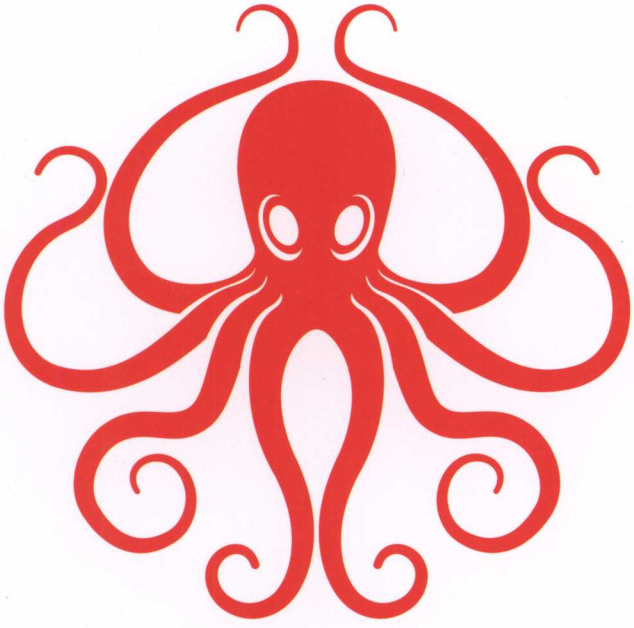
从源码角度全面解析Ceph的整体框架和各个模块的实现原理

包含作者多年开发经验，掌握分布式存储技术必备参考

HZ BOOKS 华章科技

大数据技术丛书



The Source Code Analysis of Ceph

Ceph源码分析

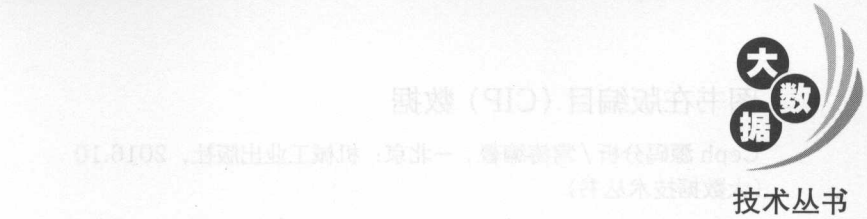
常涛◎编著

机械工业出版社
China Machine Press

作者简介

常涛

国内较早一批接触Ceph的先行者，具有多年分布式存储开发经验，曾在雅虎北京研发中心工作，后到京东、华为任职，目前在ZettaKit创业公司从事分布式存储、云计算相关技术研发。

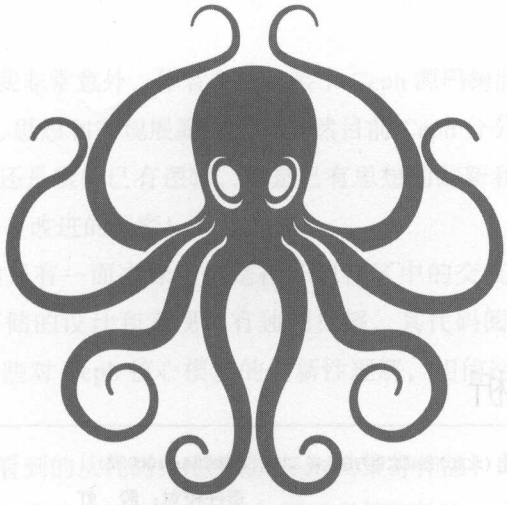


技术丛书

The Source Code Analysis of Ceph

Ceph源码分析

常涛◎编著



机械工业出版社
China Machine Press

图书在版编目（CIP）数据

Ceph 源码分析 / 常涛编著 . —北京：机械工业出版社，2016.10
（大数据技术丛书）

ISBN 978-7-111-55207-9

I. C… II. 常… III. 分布式文件系统 IV. TP316

中国版本图书馆 CIP 数据核字（2016）第 257284 号

Ceph 源码分析

出版发行：机械工业出版社（北京市西城区百万庄大街 22 号 邮政编码：100037）
责任编辑：吴 怡 责任校对：殷 虹
印 刷：北京市荣盛彩色印刷有限公司 版 次：2016 年 11 月第 1 版第 1 次印刷
开 本：186mm × 240mm 1/16 印 张：16.75
书 号：ISBN 978-7-111-55207-9 定 价：59.00 元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换
客服热线：（010）88379426 88361066 投稿热线：（010）88379604
购书热线：（010）68326294 88379649 68995259 读者信箱：hzit@hzbook.com

序　言

自从 2013 年加入 Ceph 社区以来，我一直想写一本分析 Ceph 源码的书，但是两年多来提交了数万行的代码后，我渐渐放下了这个事情。Ceph 每个月、每周都会发生巨大变化，我总是想让 Ceph 源码爱好者看到最新最棒的设计和实现，社区一线的模块维护和每周数十个代码提交集的阅读，让我很难有时间回顾和把握其他 Ceph 爱好者的疑问和需求点。

今天看到这本书让我非常意外，作者常涛把整个 Ceph 源码树肢解得恰到好处，如庖丁解牛般将 Ceph 的核心思想和实现展露出来。虽然目前 Ceph 分分钟都有新的变化，但无论是新的模块设计，还是重构已有逻辑，都是已有思想的翻新和延续，这些才是众多 Ceph 开发者能十年如一日改进的秘密！

我跟作者常涛虽然只有一面之缘，但是在开源社区中的交流已经足够成为彼此的相知。他对于分布式存储的设计和实现都有独到见解，其代码阅读和理解灵感更是超群。我在年前看到他一些对 Ceph 核心模块的创新性理解，相信这些都通过这本书展现出来了。

这本书是目前我所看到的从代码角度解读 Ceph 的最好作品，即使在全球范围内，都没有类似的书籍能够与之媲美。相信每个 Ceph 爱好者都能从这本书中找到自己心中某些疑问的解答途径。

作为 Ceph 社区的主要开发者，我也想在这里强调 Ceph 的魅力，希望每个读者都能充分感受到 Ceph 社区生机勃勃的态势。Ceph 是开源世界中存储领域的一个里程碑！在过去很难想像，从 IT 巨无霸们组成的巨大存储壁垒中能够诞生一个真正被大量用户使用并投入生产环境的开源存储项目，而 Ceph 这个开源存储项目已经成为全球众多海量存储项目的主要选择。

众所周知，在过去十年里，IT 技术领域中巨大的创新项目很多来自于开源世界，从垄断大数据的 Hadoop、Spark，到风靡全球的 Docker，都证明了开源力量推动了新技术的产生与发展。而再往以前看十年，从 Unix 到 Linux，从 Oracle 到 MySQL/PostgreSQL，从 VMWare 到 KVM，开源世界从传统商业技术继承并给用户带来更多的选择。处于开源社区一线的我欣喜地看到，在 IT 基础设施领域，越来越多的创业公司从创立之初就以开源为基石，而越来越多的商业技术公司也受益于开源，大量的复杂商业软件基于开源分布式数据库、缓存存储、中间件构建。相信开源的 Ceph 也将成为 IT 创新的驱动力。正如 Sage Weil 在 2016 Ceph Next 会议上所说，Ceph 将成为存储里的 Linux！

王豪迈，XSKY 公司 CTO

2016 年 9 月 8 日

前　言

随着云计算技术的兴起和普及，云计算基石：分布式共享存储系统受到业界的重视。Ceph 以其稳定、高可用、可扩展的特性，乘着开源云计算管理系统 OpenStack 的东风，迅速成为最热门的开源分布式存储系统。

Ceph 作为一个开源的分布式存储系统，人人都可以免费获得其源代码，并能够安装部署，但是并不等于人人都能用起来，人人都能用好。用好一个开源分布式存储系统，首先要对其架构、功能原理等方面有比较好的了解，其次要有修复漏洞的能力。这些都是在采用开源分布式存储系统时所面临的挑战。

要用好 Ceph，就必须深入了解和掌握 Ceph 源代码。Ceph 源代码的实现被公认为比较复杂，阅读难度较大。阅读 Ceph 源代码，不但需要对 C++ 语言以及 boost 库和 STL 库非常熟悉，还需要有分布式存储系统相关的基础知识以及对实现原理的深刻理解，最后还需要对 Ceph 框架和设计原理以及具体的实现细节有很好的把握。所以 Ceph 源代码的阅读是相当有挑战性的。

本着对 Ceph 源代码的浓厚兴趣以及实践工作的需要，需要对 Ceph 在源代码层级有比较深入的了解。当时笔者尽可能地搜索有关 Ceph 源代码的介绍，发现这方面的资料比较少，笔者只能自己对着 Ceph 源代码开始了比较艰辛的阅读之旅。在这个过程中，每一个小的进步都来之不易，理解一些实现细节，都需要对源代码进行反复地推敲和琢磨。自己在阅读的过程中，特别希望有人能够帮助理清整体代码的思路，能够解答一下关键的实现细节。本书就是秉承这样一个简单的目标，希望指引和帮助广大 Ceph 爱好者更好地理解和掌握 Ceph 源代码。

本书面向热爱 Ceph 的开发者，想深入了解 Ceph 原理的高级运维人员，想基于 Ceph 做优化和定制的开发人员，以及想对社区提交代码的研究人员。官网上有比较详细的介

绍 Ceph 安装部署以及操作相关的知识，希望阅读本书的人能够自己动手实践，对 Ceph 进一步了解。本书基于目前最新的 Ceph 10.2.1 版本进行分析。

本书着重介绍 Ceph 的整体框架和各个实现模块的实现原理，对核心源代码进行分析，包括一些关键的实现细节。存储系统的实现都是围绕数据以及对数据的操作来展开，只要理解核心的数据结构，以及数据结构的相关操作就可以大致了解核心的实现和功能。本书的写作思路是先介绍框架和原理，其次介绍相关的数据结构，最后基于数据结构，介绍相关的操作实现流程。

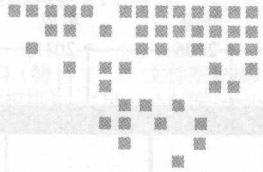
最后感谢一起工作过的同事们，同他们在 Ceph 技术上进行交流沟通并加以验证实践，使我受益匪浅。感谢机械工业出版社的编辑吴怡对本书出版所做的努力，以及不断提出的宝贵意见。感谢我的妻子孙盛南女士在我写作期间默默的付出，对本书的写作提供了坚强的后盾。

由于 Ceph 源代码比较多，也比较复杂，写作的时间比较紧，加上个人的水平有限，错误和疏漏在所难免，恳请读者批评指正。有任何的意见和建议都可发送到我的邮箱 changtao381@163.com，欢迎读者与我交流 Ceph 相关的任何问题。

常涛

2016 年 6 月于北京

目　录



第 1 章 Chapter 1

Ceph 整体架构

本章从比较高的层次对 Ceph 的发展历史、Ceph 的设计目标、整体架构进行简要介绍。其次介绍 Ceph 的三种对外接口：块存储、对象存储、文件存储。还介绍 Ceph 的存储基石 RADOS 系统的一些基本概念、各个模块组成和功能。最后介绍了对象的寻址过程和数据读写的原理，以及 RADOS 实现的数据服务等。

1.1 Ceph 的发展历程

Ceph 项目起源于其创始人 Sage Weil 在加州大学 Santa Cruz 分校攻读博士期间的研究课题。项目的起始时间为 2004 年，在 2006 年基于开源协议开源了 Ceph 的源代码。Sage Weil 也相应成立了 Inktank 公司专注于 Ceph 的研发。在 2014 年 5 月，该公司被 Red Hat 收购。Ceph 项目的发展历程如图 1-1 所示。

2012 年，Ceph 发布了第一个稳定版本。2014 年 10 月，Ceph 开发团队发布了 Ceph 的第七个稳定版本 Giant。到目前为止，社区平均每三个月发布一个稳定版本，目前的最新版本为 10.2.1。

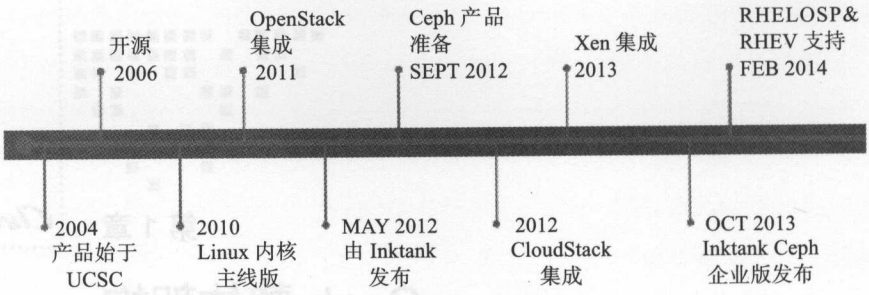


图 1-1 Ceph 的发展历程

1.2 Ceph 的设计目标

Ceph 的设计目标是采用商用硬件（Commodity Hardware）来构建大规模的、具有高可用性、高可扩展性、高性能的分布式存储系统。

商用硬件一般指标准的 x86 服务器，相对于专用硬件，性能和可靠性较差，但由于价格相对低廉，可以通过集群优势来发挥高性能，通过软件的设计解决高可用性和可扩展性。标准化的硬件可以极大地方便管理，且集群的灵活性可以应对多种应用场景。

系统的高可用性指的是系统某个部件失效后，系统依然可以提供正常服务的能力。一般用设备部件和数据的冗余来提高可用性。Ceph 通过数据多副本、纠删码来提供数据的冗余。

高可扩展性是指系统可以灵活地应对集群的伸缩。一般指两个方面，一方面指集群的容量可以伸缩，集群可以任意地添加和删除存储节点和存储设备；另一方面指系统的性能随集群的增加而线性增加。

大规模集群环境下，要求 Ceph 存储系统的规模可以扩展到成千上万个节点。当集群规模达到一定程度时，系统在数据恢复、数据迁移、节点监测等方面会产生一系列富有挑战性的问题。

1.3 Ceph 基本架构图

Ceph 的整体架构由三个层次组成：最底层也是最核心的部分是 RADOS 对象存储系统。

第二层是 librados 库层；最上层对应着 Ceph 不同形式的存储接口实现，架构如图 1-2 所示。

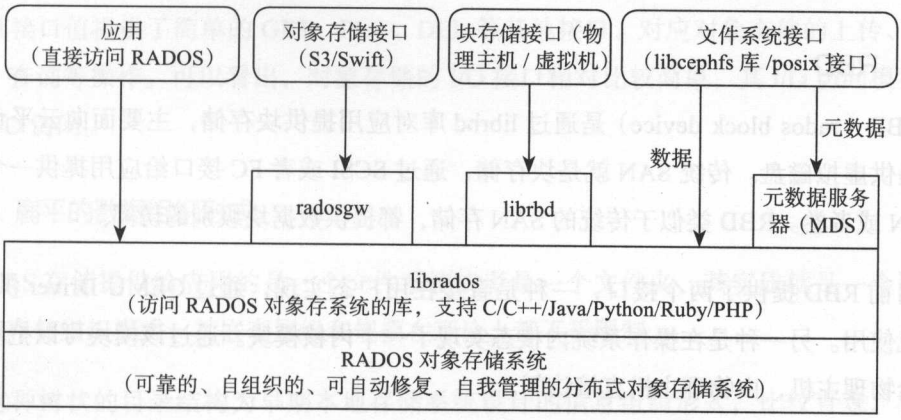


图 1-2　Ceph 基本架构图

Ceph 的整体架构大致如下：

- 最底层基于 RADOS(reliable, autonomous, distributed object store)，它是一个可靠的、自组织的、可自动修复、自我管理的分布式对象存储系统。其内部包括 ceph-osd 后台服务进程和 ceph-mon 监控进程。
- 中间层 librados 库用于本地或者远程通过网络访问 RADOS 对象存储系统。它支持多种语言，目前支持 C/C++ 语言、Java、Python、Ruby 和 PHP 语言的接口。
- 最上层面向应用提供 3 种不同的存储接口：
  - 块存储接口，通过 librbd 库提供了块存储访问接口。它可以为虚拟机提供虚拟磁盘，或者通过内核映射为物理主机提供磁盘空间。
  - 对象存储接口，目前提供了两种类型的 API，一种是和 AWS 的 S3 接口兼容的 API，另一种是和 OpenStack 的 Swift 对象接口兼容的 API。
  - 文件系统接口，目前提供两种接口，一种是标准的 posix 接口，另一种通过 libcephfs 库提供文件系统访问接口。文件系统的元数据服务器 MDS 用于提供元数据访问。数据直接通过 librados 库访问。

1.4　Ceph 客户端接口

Ceph 的设计初衷是成为一个分布式文件系统，但随着云计算的大量应用，最终变成

支持三种形式的存储：块存储、对象存储、文件系统，下面介绍它们之间的区别。

1.4.1 RBD

RBD（rados block device）是通过 librbd 库对应用提供块存储，主要面向云平台的虚拟机提供虚拟磁盘。传统 SAN 就是块存储，通过 SCSI 或者 FC 接口给应用提供一个独立的 LUN 或者卷。RBD 类似于传统的 SAN 存储，都提供数据块级别的访问。

目前 RBD 提供了两个接口，一种是直接在用户态实现，通过 QEMU Driver 供 KVM 虚拟机使用。另一种是在操作系统内核态实现了一个内核模块。通过该模块可以把块设备映射给物理主机，由物理主机直接访问。

块存储用作虚拟机的硬盘，其对 I/O 的要求和传统的物理硬盘类似。一个硬盘应该是能面向通用需求的，既能应付大文件读写，也能处理好小文件读写。也就是说，块存储既需要有较好的随机 I/O，又要求有较好的顺序 I/O，而且对延迟有比较严格的要求。

1.4.2 CephFS

CephFS 通过在 RADOS 基础之上增加了 MDS（Metadata Server）来提供文件存储。它提供了 libcephfs 库和标准的 POSIX 文件接口。CephFS 类似于传统的 NAS 存储，通过 NFS 或者 CIFS 协议提供文件系统或者文件目录服务。

Ceph 最初的设计为分布式文件系统，其通过动态子树的算法实现了多元数据服务器，但是由于实现复杂，目前还远远不能使用。目前可用于生产环境的是最新 Jewel 版本的 CephFS 为主从模式（Master-Slave）的元数据服务器。

1.4.3 RadosGW

RadosGW 基于 librados 提供了和 Amazon S3 接口以及 OpenStack Swift 接口兼容的对象存储接口。可将其简单地理解为提供基本文件（或者对象）的上传和下载的需求，它有两个特点：

- 提供 RESTful Web API 接口。
- 它采用扁平的数据组织形式。

1. RESTful 的存储接口

其接口值提供了简单的 GET、PUT、DEL 等其他接口，对应对象文件的上传、下载、删除、查询等操作。可以看出，对象存储的 I/O 接口相对比较简单，其 I/O 访问模型都是顺序 I/O 访问。

2. 扁平的数据组织形式

NAS 存储提供给应用的是一个文件系统或者是一个文件夹，其实质就是一个层级化的树状存储组织模式，其嵌套层级和规模在理论上都不受限制。

这种树状的目录结构为早期本地存储系统设计的信息组织形式，比较直观，容易理解。但是随着存储系统规模的不断扩大，特别是到了云存储时代，其难以大规模扩展的缺点就暴露了出来。相比于 NAS 存储，对象存储放弃了目录树结构，采用了扁平化组织形式（一般为三级组织结构），这有利于实现近乎无限的容量扩展。它使用业界标准互联网协议，更加符合面向云服务的存储、归档和备份需求。

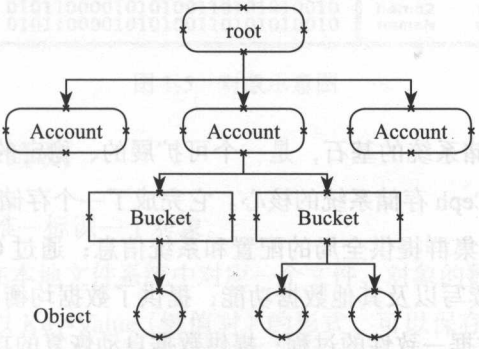


图 1-3　Amazon S3 的对象存储结构

由于 Amazon 在云存储领域的影响力，Amazon 的 S3 接口已经成为事实上的对象存储的标准接口。如图 1-3 所示，其接口分三级存储：Account/Bucket/Object（账户 / 桶 / 对象）。一个 Account 可以看作一个用户（租户），其下可以包含若干个的 Bucket，一个 Bucket 可以拥有若干对象，其数量在理论上都不受限制。

在云计算领域，OpenStack 已经成为广泛采用的云计算管理系统，OpenStack 的对象存储接口 Swift 也成为广泛采用的接口，如图 1-4 所示，其也采用分三级存储：Account/

Container/Object（账户 / 容器 / 对象），每层节点数均没有限制。可以看出，Swift 接口和 S3 类似，Swift 的 Container 对应 S3 的 Bucket 概念。Swift 接口和 S3 接口没有太大的区别，但是一些管理接口会有一些差别。

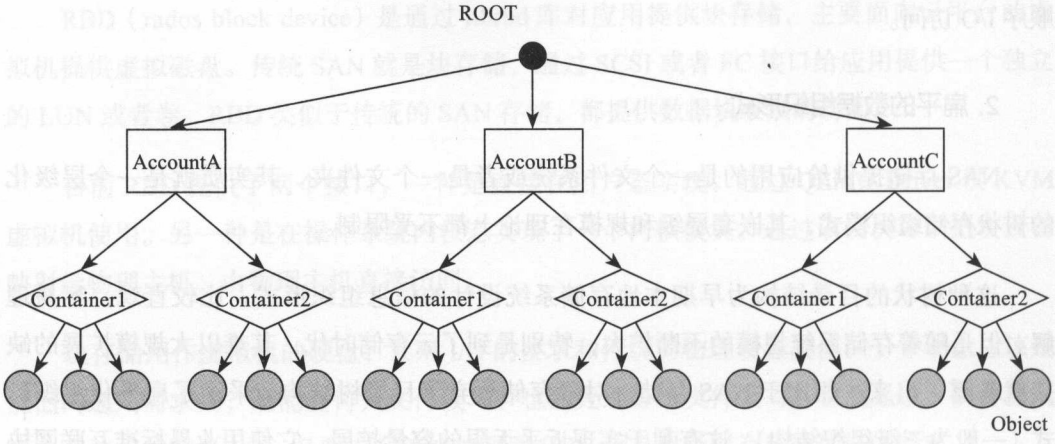


图 1-4 OpenStack Swift 对象存储结构

1.5 RADOS

RADOS 是 Ceph 存储系统的基石，是一个可扩展的、稳定的、自我管理的、自我修复的对象存储系统，是 Ceph 存储系统的核心。它完成了一个存储系统的核心功能，包括：Monitor 模块为整个存储集群提供全局的配置和系统信息；通过 CRUSH 算法实现对象的寻址过程；完成对象的读写以及其他数据功能；提供了数据均衡功能；通过 Peering 过程完成一个 PG 内存达成数据一致性的过程；提供数据自动恢复的功能；提供克隆和快照功能；实现了对象分层存储的功能；实现了数据一致性检查工具 Scrub。下面分别对上述基本功能做简要的介绍。

1.5.1 Monitor

Monitor 是一个独立部署的 daemon 进程。通过组成 Monitor 集群来保证自己的高可用。Monitor 集群通过 Paxos 算法实现了自己数据的一致性。它提供了整个存储系统的节点信息等全局的配置信息。

Cluster Map 保存了系统的全局信息，主要包括：

- Monitor Map
  - 包括集群的 fsid
  - 所有 Monitor 的地址和端口
  - current epoch
- OSD Map：所有 OSD 的列表，和 OSD 的状态等。
- MDS Map：所有的 MDS 的列表和状态。

1.5.2　对象存储

这里所说的对象是指 RADOS 对象，要和 RadosGW 的 S3 或者 Swift 接口的对象存储区分开来。对象是数据存储的基本单元，一般默认 4MB 大小。图 1-5 就是一个对象的示意图。

ID	Binary Data	Metadata
1234	01010101010100110101010010 01011000010101001101010010 01011000010101001101010010	name1　value1 name2　value2 nameN　valueN

图 1-5　对象示意图

一个对象由三个部分组成：

- 对象标志（ID），唯一标识一个对象。
- 对象的数据，其在本地文件系统中对应一个文件，对象的数据就保存在文件中。
- 对象的元数据，以 Key-Value（键值对）的形式，可以保存在文件对应的扩展属性中。由于本地文件系统的扩展属性能保存的数据量有限制，RADOS 增加了另一种方式：以 Leveldb 等的本地 KV 存储系统来保存对象的元数据。

1.5.3　pool 和 PG 的概念

pool 是一个抽象的存储池。它规定了数据冗余的类型以及对应的副本分布策略。目前实现了两种 pool 类型：replicated 类型和 Erasure Code 类型。一个 pool 由多个 PG 构成。

PG（placement group）从名字可理解为一个放置策略组，它是对象的集合，该集合里

的所有对象都具有相同的放置策略：对象的副本都分布在相同的 OSD 列表上。一个对象只能属于一个 PG，一个 PG 对应于放置在其上的 OSD 列表。一个 OSD 上可以分布多个 PG。

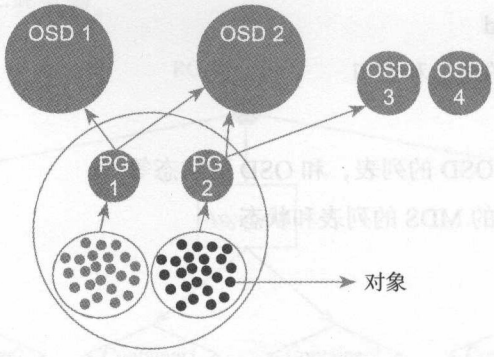


图 1-6　PG 的概念示意图

PG 的概念如图 1-6 所示，其中：

- PG1 和 PG2 都属于同一个 pool，所以都是副本类型，并且都是两副本。
- PG1 和 PG2 里都包含许多对象，PG1 上的所有对象，主从副本分布在 OSD1 和 OSD2 上，PG2 上的所有对象的主从副本分布在 OSD2 和 OSD3 上。
- 一个对象只能属于一个 PG，一个 PG 包含多个对象。
- 一个 PG 的副本分布在对应的 OSD 列表中。在一个 OSD 上可以分布多个 PG。示例中 PG1 和 PG2 的从副本都分布在 OSD2 上。

1.5.4　对象寻址过程

对象寻址过程指的是查找对象在集群中分布的位置信息，过程分为两步：

1）对象到 PG 的映射。这个过程是静态 hash 映射（加入 pg split 后实际变成了动态 hash 映射方式），通过对 object_id，计算出 hash 值，用该 pool 的 PG 的总数量 pg_num 对 hash 值取模，就可以获得该对象所在的 PG 的 id 号：

```
pg_id = hash( object_id ) % pg_num
```

2）PG 到 OSD 列表映射。这是指 PG 上对象的副本如何分布在 OSD 上。它使用 Ceph 自己创新的 CRUSH 算法来实现，本质上是一个伪随机分布算法。

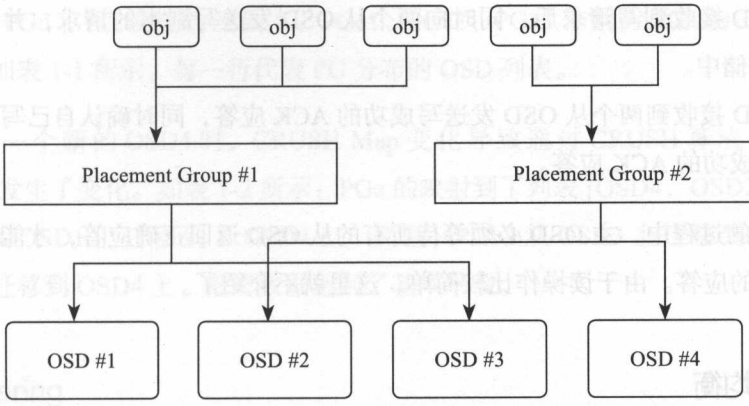


图 1-7　对象寻址过程

如图 1-7 所示的对象寻址过程：

1）通过 hash 取模后计算，前三个对象分布在 PG1 上，后两个对象分布在 PG2 上。

2）PG1 通过 CRUSH 算法，计算出 PG1 分布在 OSD1、OSD3 上；PG2 通过 CRUSH 算法分布在 OSD2 和 OSD4 上。

1.5.5　数据读写过程

Ceph 的数据写操作如图 1-8 所示，其过程如下：

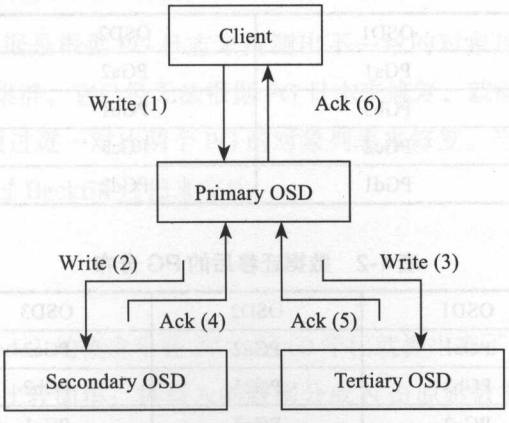


图 1-8　读写过程

1）Client 向该 PG 所在的主 OSD 发送写请求。

2）主 OSD 接收到写请求后，同时向两个从 OSD 发送写副本的请求，并同时写入主 OSD 的本地存储中。

3）主 OSD 接收到两个从 OSD 发送写成功的 ACK 应答，同时确认自己写成功，就向客户端返回写成功的 ACK 应答。

在写操作的过程中，主 OSD 必须等待所有的从 OSD 返回正确应答，才能向客户端返回写操作成功的应答。由于读操作比较简单，这里就不介绍了。

1.5.6 数据均衡

当在集群中新添加一个 OSD 存储设备时，整个集群会发生数据的迁移，使得数据分布达到均衡。Ceph 数据迁移的基本单位是 PG，即数据迁移是将 PG 中的所有对象作为一个整体来迁移。

迁移触发的流程为：当新加入一个 OSD 时，会改变系统的 CRUSH Map，从而引起对象寻址过程中的第二步，PG 到 OSD 列表的映射发生了变化，从而引发数据的迁移。

举例来说明数据迁移过程，表 1-1 是数据迁移前的 PG 分布，表 1-2 是数据迁移后的 PG 分布。

表 1-1 数据迁移前的 PG 分布

	OSD1	OSD2	OSD3
PGa	PGa1	PGa2	PGa3
PGb	PGb3	PGb1	PGb2
PGc	PGc2	PGc3	PGc1
PGd	PGd1	PGd2	PGd3

表 1-2 数据迁移后的 PG 分布

	OSD1	OSD2	OSD3	OSD4
PGa	~~PGa1~~	PGa2	PGa3	PGa1
PGb	PGb3	PGb1	~~PGb2~~	PGb2
PGc	PGc2	~~PGc3~~	PGc1	PGc3
PGd	PGd1	PGd2	PGd3	

当前系统有 3 个 OSD，分布为 OSD1、OSD2、OSD3；系统有 4 个 PG，分布为 PGa、

PGb、PGc、PGd；PG 设置为三副本：PGa1、PGa2、PGa3 分别为 PGa 的三个副本。PG 的所有分布如表 1-1 所示，每一行代表 PG 分布的 OSD 列表。

当添加一个新的 OSD4 时，CRUSH Map 变化导致通过 CRUSH 算法来计算 PG 到 OSD 的分布发生了变化。如表 1-2 所示：PGa 的映射到了列表 [OSD4，OSD2，OSD3] 上，导致 PGa1 从 OSD1 上迁移到了 OSD4 上。同理，PGb2 从 OSD3 上迁移到 OSD4，PGc3 从 OSD2 上迁移到 OSD4 上，最终数据达到了基本平衡。

1.5.7　Peering

当 OSD 启动，或者某个 OSD 失效时，该 OSD 上的主 PG 会发起一个 Peering 的过程。Ceph 的 Peering 过程是指一个 PG 内的所有副本通过 PG 日志来达成数据一致的过程。当 Peering 完成之后，该 PG 就可以对外提供读写服务了。此时 PG 的某些对象可能处于数据不一致的状态，其被标记出来，需要恢复。在写操作的过程中，遇到处于不一致的数据对象需要恢复的话，则需要等待，系统优先恢复该对象后，才能继续完成写操作。

1.5.8　Recovery 和 Backfill

Ceph 的 Recovery 过程是根据在 Peering 的过程中产生的、依据 PG 日志推算出的不一致对象列表来修复其他副本上的数据。

Recovery 过程的依据是根据 PG 日志来推测出不一致的对象加以修复。当某个 OSD 长时间失效后重新加入集群，它已经无法根据 PG 日志来修复，就需要执行 Backfill（回填）过程。Backfill 过程是通过逐一对比两个 PG 的对象列表来修复。当新加入一个 OSD 产生了数据迁移，也需要通过 Backfill 过程来完成。

1.5.9　纠删码

纠删码（Erasure Code）的概念早在 20 世纪 60 年代就提出来了，最近几年被广泛应用在存储领域。它的原理比较简单：将写入的数据分成 N 份原始数据块，通过这 N 份原始数据块计算出 M 份效验数据块，N+M 份数据块可以分别保存在不同的设备或者节点中。可以允许最多 M 个数据块失效，通过 N+M 份中的任意 N 份数据，就还原出其他数据块。

目前 Ceph 对纠删码（EC）的支持还比较有限。RBD 目前不能直接支持纠删码（EC）模式。其或者应用在对象存储 radosgw 中，或者作为 Cache Tier 的二层存储。其中的原因和具体实现都将在后面的章节详细介绍。

1.5.10 快照和克隆

快照（snapshot）就是一个存储设备在某一时刻的全部只读镜像。克隆（clone）是在某一时刻的全部可写镜像。快照和克隆的区别在于快照只能读，而克隆可写。

RADOS 对象存储系统本身支持 Copy-on-Write 方式的快照机制。基于这个机制，Ceph 可以实现两种类型的快照，一种是 pool 级别的快照，给整个 pool 中的所有对象统一做快照操作。另一种就是用户自己定义的快照实现，这需要客户端配合实现一些快照机制。RBD 的快照实现就属于后者。

RBD 的克隆实现是在基于 RBD 的快照基础上，在客户端 librbd 上实现了 Copy-on-Write（cow）克隆机制。

1.5.11 Cache Tier

RADOS 实现了以 pool 为基础的自动分层存储机制。它在第一层可以设置 cache pool，其为高速存储设备（例如 SSD 设备）。第二层为 data pool，使用大容量低速存储设备（如 HDD 设备）可以使用 EC 模式来降低存储空间。通过 Cache Tier，可以提高关键数据或者热点数据的性能，同时降低存储开销。

Cache Tier 的结构如图 1-9 所示，说明如下：

- ❑ Ceph Client 对于 Cache 层是透明的。
- ❑ 类 Objecter 负责请求是发给 Cache Tier 层，还是发给 Storage Tier 层。
- ❑ Cache Tier 层为高速 I/O 层，保存热点数据，或称为活跃的数据。
- ❑ Storage Tier 层为慢速层，保存非活跃的数据。
- ❑ 在 Cache Tier 层和 Storage Tier 层之间，数据根据活跃度自动地迁移。

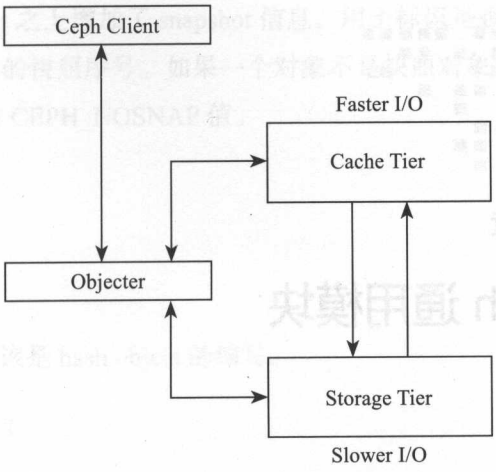


图 1-9　Cache Tier 结构图

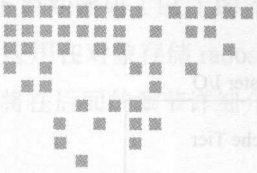
1.5.12　Scrub

Scrub 机制用于系统检查数据的一致性。它通过在后台定期（默认每天一次）扫描，比较一个 PG 内的对象分别在其他 OSD 上的各个副本的元数据和数据来检查是否一致。根据扫描的内容分为两种，第一种是只比较对象各个副本的元数据，它代价比较小，扫描比较高效，对系统影响比较小。另一种扫描称为 deep scrub，它需要进一步比较副本的数据内容检查数据是否一致。

1.6　本章小结

本章介绍了 Ceph 的系统架构，通过本章，可以对 Ceph 的基本架构和各个模块的组件有了整体的了解，并对一些基本概念及读写的原理、各个数据功能模块有了大致了解。

本章介绍的 Ceph 客户端的基本概念，在第 5 章会详细介绍到。本章介绍的对象寻址的 CRUSH 算法将会在第 4 章详细介绍到。在本章介绍的本地对象存储将会在第 7 章详细介绍到。本章介绍的数据读写流程，将会在第 6 章详细介绍。本章介绍的纠删码将在第 8 章中详细介绍。本章介绍的快照和克隆将在第 9 章详细介绍。本章介绍的 Ceph Peering 的过程将会在第 10 章详细介绍。本章介绍的数据恢复和回填将会在第 11 章介绍到。本章介绍的 Ceph Srcub 机制将会在第 12 章详细介绍。本章介绍的 Cache Tier 将在第 13 章详细介绍。



Chapter 2 第 2 章

Ceph 通用模块

本章介绍 Ceph 源代码通用库中的一些比较关键而又比较复杂的数据结构。Object 和 Buffer 相关的数据结构是普遍使用的。线程池 ThreadPool 可以提高消息处理的并发能力。Finisher 提供了异步操作时来执行回调函数。Throttle 在系统的各个模块各个环节都可以看到，它用来限制系统的请求，避免瞬时大量突发请求对系统的冲击。SafteTimer 提供了定时器，为超时和定时任务等提供了相应的机制。理解这些数据结构，能够更好理解后面章节的相关内容。

2.1 Object

对象 Object 是默认为 4MB 大小的数据块。一个对象就对应本地文件系统中的一个文件。在代码实现中，有 object、sobject、hobject、ghobject 等不同的类。

结构 object_t 对应本地文件系统的一个文件，name 就是对象名：

```
struct object_t {
  string name;
  ......
}
```

sobject_t 在 object_t 之上增加了 snapshot 信息，用于标识是否是快照对象。数据成员 snap 为快照对象的对应的快照序号。如果一个对象不是快照对象（也就是 head 对象），那么 snap 字段就被设置为 CEPH_NOSNAP 值。

```
struct sobject_t {
  object_t oid;
  snapid_t snap;
  ......
}
```

hobject_t 是名字应该是 hash object 的缩写。

```
struct hobject_t {
  object_t oid;
  snapid_t snap;
private:
  uint32_t hash;
  bool max;
  uint32_t nibblewise_key_cache;
  uint32_t hash_reverse_bits;
  ......
public:
  int64_t pool;
  string  nspace;

private:
  string key;
  ......
}
```

其在 sobject_t 的基础上增加了一些字段：

- int64_t pool：所在的 pool 的 id。
- string nspace：nspace 一般为空，它用于标识特殊的对象。
- string key：对象的特殊标记。
- string hash：hash 和 key 不能同时设置，hash 值一般设置为就是 pg 的 id 值。

ghobject_t 在对象 hobject_t 的基础上，添加了 generation 字段和 shard_id 字段，这个用于 ErasureCode 模式下的 PG：

- shard_id 用于标识对象所在的 osd 在 EC 类型的 PG 中的序号，对应 EC 来说，每个 osd 在 PG 中的序号在数据恢复时非常关键。如果是 Replicate 类型的 PG，那么字

段就设置为 NO_SHARD(-1)，该字段对于 replicate 是没用。

- generation 用于记录对象的版本号。当 PG 为 EC 时，写操作需要区分写前后两个版本的 object，写操作保存对象的上一个版本（generation）的对象，当 EC 写失败时，可以 rollback 到上一个版本。

```
struct ghobject_t {
  hobject_t hobj;
  gen_t generation;
  shard_id_t shard_id;
  bool max;

public:
  static const gen_t NO_GEN = UINT64_MAX;
  ......
}
```

2.2 Buffer

Buffer 就是一个命名空间，在这个命名空间下定义了 Buffer 相关的数据结构，这些数据结构在 Ceph 的源代码中广泛使用。下面介绍的 buffer::raw 类是基础类，其子类完成了 Buffer 数据空间的分配，buffer::ptr 类实现了 Buffer 内部的一段数据，buffer::list 封装了多个数据段。

2.2.1 buffer::raw

类 buffer::raw 是一个原始的数据 Buffer，在其基础之上添加了长度、引用计数和额外的 crc 校验信息，结构如下：

```
class buffer::raw {
  public:
    char *data;        // 数据指针
    unsigned len;      // 数据长度
    atomic_t nref;     // 引用计数

    mutable RWLock crc_lock;    // 读写锁，保护 crc_map
    map<pair<size_t, size_t>, pair<uint32_t, uint32_t> > crc_map;
     //crc 校验信息，第一个 pair 为数据段的起始和结束（from,to），第二个 pair 是 crc32 校验
       码，pair 的第一字段为 base crc32 校验码，第二个字段为加上数据段后计算出的 crc32 校验码。
......
}
```

下列类都继承了 buffer::raw，实现了 data 对应内存空间的申请：

- 类 raw_malloc 实现了用 malloc 函数分配内存空间的功能。
- 类 class buffer::raw_mmap_pages 实现了通过 mmap 来把内存匿名映射到进程的地址空间。
- 类 class buffer::raw_posix_aligned 调用了函数 posix_memalign 来申请内存地址对齐的内存空间。
- 类 class buffer::raw_hack_aligned 是在系统不支持内存对齐申请的情况下自己实现了内存地址的对齐。
- 类 class buffer::raw_pipe 实现了 pipe 做为 Buffer 的内存空间。
- 类 class buffer::raw_char 使用了 C++ 的 new 操作符来申请内存空间。

2.2.2　buffer::ptr

类 buffer::ptr 就是对于 buffer::raw 的一个部分数据段。结构如下：

```
class CEPH_BUFFER_API ptr {
  raw *_raw;
  unsigned _off, _len;
  ......
}
```

ptr 是 raw 里的一个任意的数据段，_off 是在 _raw 里的偏移量，_len 是 ptr 的长度。raw 和 ptr 的示意图如图 2-1 所示。

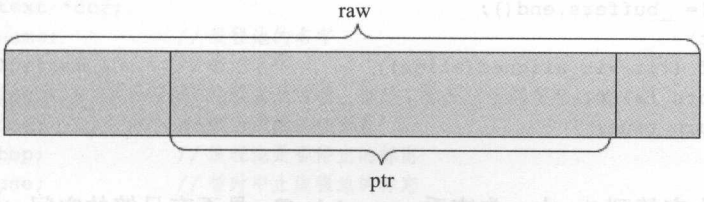


图 2-1　raw 和 ptr 示意图

2.2.3　buffer::list

类 buffer::list 是一个使用广泛的类，它是多个 buffer::ptr 的列表，也就是多个内存数据段的列表。结构如下：

```
class CEPH_BUFFER_API list {
  std::list<ptr> _buffers; //所有的 ptr
  unsigned _len;           //所有的 ptr 的数据总长度
  unsigned _memcopy_count; //当调用函数 rebuild 用来内存对齐时，需要内存拷贝的数据量
  ptr append_buffer;       //当有小的数据就添加到这个 buffer 里
  mutable iterator last_p; //访问 list 的迭代器
  ......
}
```

buffer::list 的重要的操作如下所示。

- 添加一个 ptr 到 list 的头部：

```
void push_front(ptr& bp) {
  if (bp.length() == 0)
    return;
  _buffers.push_front(bp);
  _len += bp.length();
}
```

- 添加一个 raw 到 list 头部中，先构造一个 ptr，后添加 list 中：

```
void push_front(raw *r) {
  ptr bp(r);
  push_front(bp);
}
```

- 判断内存是否以参数 align 对齐，每一个 ptr 都必须以 align 对齐：

```
bool buffer::list::is_aligned(unsigned align) const
{
  for (std::list<ptr>::const_iterator it = _buffers.begin();
    it != _buffers.end();
    ++it)
      if (!it->is_aligned(align))
    return false;
    return true;
}
```

- 添加一个字符到 list 中，先查看 append_buffer 是否有足够的空间，如果没有，就新申请一个 4KB 大小的空间：

```
void buffer::list::append(char c)
{
  // 检查当前的 append_buffer 是否有足够的空间
  unsigned gap = append_buffer.unused_tail_length();
```

```
    if (!gap) {
      // 如果没有空间，就申请一个 append_buffer!
    append_buffer = create_aligned(CEPH_BUFFER_APPEND_SIZE,
                                   CEPH_BUFFER_APPEND_SIZE);
      append_buffer.set_length(0);   // 到目前为止，没有用到
    }
      append(append_buffer, append_buffer.append(c) - 1, 1);
      // 把该数据段添加到 append_buffer 中
  }
```

- 内存对齐：有些情况下，需要内存地址对齐，例如当以 directIO 方式写入数据至磁盘时，需要内存地址按内存页面大小（page）对齐，也即 buffer::list 的内存地址都需按 page 对齐。函数 rebuild 用来完成对齐的功能。其实现的方法也比较简单，检查没有对齐的 ptr，申请一块新对齐的内存，把数据拷贝过去，释放内存空间就可以了。
- buffer::list 还集成了其他额外的一些功能：
  - 把数据写入文件或从文件读取数据的功能。
  - 计算数据的 crc32 校验。

2.3　线程池

线程池（ThreadPool）在分布式存储系统的实现中是必不可少的，在 Ceph 的代码中广泛用到。Ceph 中线程池的实现也比较复杂，结构如下：

```
class ThreadPool : public md_config_obs_t {
  CephContext *cct;
  string name;              // 线程池的名字
  string lockname;          // 锁的名字
  Mutex _lock;              // 线程互斥的锁，也是工作队列访问互斥的锁
  Cond _cond;               // 锁对应的条件变量
  bool _stop;               // 线程池是否停止的标志
  int _pause;               // 暂时中止线程池的标志
  int _draining;
  Cond _wait_cond;
  int ioprio_class, ioprio_priority;

  vector<WorkQueue_*> work_queues;      // 工作队列
  int last_work_queue;                  // 最后访问的工作队列

  set<WorkThread*> _threads;            // 线程池中的工作线程
```

```
  list<WorkThread*> _old_threads;        // 等待进 joined 操作的线程
  int processing;
}
```

类 ThreadPool 里包函一些比较重要的数据成员：

- 工作线程集合 _threads。
- 等待 Join 操作的旧线程集合 _old_threads。
- 工作队列集合，保存所有要处理的任务。一般情况下，一个工作队列对应一个类型的处理任务，一个线程池对应一个工作队列，专门用于处理该类型的任务。如果是后台任务，又不紧急，就可以将多个工作队列放置到一个线程池里，该线程池可以处理不同类型的任务。

线程池的实现主要包括：线程池的启动过程，线程池对应的工作队列的管理，线程池对应的执行函数如何执行任务。下面分别介绍这些实现，然后介绍一些 Ceph 线程池实现的超时检查功能，最后介绍 ShardedThreadpool 的实现原理。

2.3.1 线程池的启动

函数 ThreadPool::start() 用来启动线程池，其在加锁的情况下，调用函数 start_threads，该函数检查当前线程数，如果小于配置的线程池，就创建新的工作线程。

2.3.2 工作队列

工作队列（WorkQueue）定义了线程池要处理的任务。任务类型在模板参数中指定。在构造函数里，就把自己加入到线程池的工作队列集合中：

```
template<class T>
class WorkQueue : public WorkQueue_ {
  ThreadPool *pool;
  WorkQueue(string n, time_t ti, time_t sti, ThreadPool* p) : WorkQueue_
          (n, ti, sti), pool(p) {
    pool->add_work_queue(this);
  }
  ......
}
```

WorkQueue 实现了一部分功能：进队列和出队列，以及加锁，并用通过条件变量通知

相应的处理线程：

```
bool queue(T *item) {
  pool->_lock.Lock();
  bool r = _enqueue(item);
  pool->_cond.SignalOne();
  pool->_lock.Unlock();
  return r;
  }
  void dequeue(T *item) {
    pool->_lock.Lock();
    _dequeue(item);
    pool->_lock.Unlock();
  }
  void clear() {
    pool->_lock.Lock();
    _clear();
    pool->_lock.Unlock();
}
```

还有一部分功能，需要使用者自己定义。需要自己定义实现保存任务的容器，添加和删除的方法，以及如何处理任务的方法：

```
virtual bool _enqueue(T *) = 0;
    //从提交的任务中去除一个项
virtual void _dequeue(T *) = 0;
    //去除一个项并返回原始指针
virtual T *_dequeue() = 0;
virtual void _process(T *t) { assert(0); }
virtual void _process(T *t, TPHandle &) {
      _process(t);
```

2.3.3　线程池的执行函数

函数 worker 为线程池的执行函数：

```
void ThreadPool::worker(WorkThread *wt)
```

其处理过程如下：

1）首先检查 _stop 标志，确保线程池没有关闭。

2）调用函数 join_old_threads 把旧的工作线程释放掉。检查如果线程数量大于配置的数量 _num_threads，就把当前线程从线程集合中删除，并加入 _old_threads 队列中，并退出循环。

3）如果线程池没有暂时中止，并且 work_queues 不为空，就从 last_work_queue 开始，遍历每一个工作队列，如果工作队列不为空，就取出一个 item，调用工作队列的处理函数做处理。

2.3.4 超时检查

TPHandle 是一个有意思的事情。每次线程函数执行时，都会设置一个 grace 超时时间，当线程执行超过该时间，就认为是 unhealthy 的状态。当执行时间超过 suicide_grace 时，OSD 就会产生断言而导致自杀，代码如下：

```
struct heartbeat_handle_d {
  const std::string name;
  atomic_t timeout, suicide_timeout;
  time_t grace, suicide_grace;
  std::list<heartbeat_handle_d*>::iterator list_item;
}

class TPHandle {
  friend class ThreadPool;
  CephContext *cct;
  heartbeat_handle_d *hb;    // 心跳
  time_t grace;              // 超时
  time_t suicide_grace;      // 自杀的超时时间
}
```

结构 heartbeat_handle_d 记录了相关信息，并把该结构添加到 HeartbeatMap 的系统链表中保存。OSD 会有一个定时器，定时检查是否超时。

2.3.5 ShardedThreadPool

这里简单介绍一个 ShardedThreadPool。在之前的介绍中，ThreadPool 实现的线程池，其每个线程都有机会处理工作队列的任意一个任务。这就会导致一个问题，如果任务之间有互斥性，那么正在处理该任务的两个线程有一个必须等待另一个处理完成后才能处理，从而导致线程的阻塞，性能下降。

例 2-1 如表 2-1 所示，线程 Thread1 和 Thread2 分别正在处理 Job1 和 Job2。

由于 Job1 和 Job2 的关联性，二者不能并发执行，只能顺序执行，二者之间用一个互斥锁来控制。如果 Thread1 先获得互斥锁就先执行，Thread2 必须等待，直到 Thread1 执行

完 Job1 后释放了该互斥锁，Thread2 获得该互斥锁后才能执行 Job2。显然，这种任务的调度方式应对这种不能完全并行的任务是有缺陷的。实际上 Thread2 可以去执行其他任务，比如 Job5。Job1 和 Job2 既然是顺序的，就都可以交给 Thread1 执行。

表 2-1　ThreadPool 的处理模型示列

线程\任务	Job1	Job2	Job3	Job4	Job5
Thread1	*				
Thread2		*			
Thread3				*	
Thread4			*		

因此，引入了 Sharded ThreadPool 进行管理。ShardedThreadPool 对上述的任务调度方式做了改进，其在线程的执行函数里，添加了表示线程的 thread_index：

```
void shardedthreadpool_worker(uint32_t thread_index);
```

具体如何实现 Shard 方式，还需要使用者自己去实现。其基本的思想就是：每个线程对应一个任务队列，所有需要顺序执行的任务都放在同一个线程的任务队列里，全部由该线程执行。

2.4　Finisher

类 Finisher 用来完成回调函数 Context 的执行，其内部有一个 FinisherThread 线程来用于执行 Context 回调函数：

```
class Finisher {
  ......
  vector<Context*> finisher_queue;
                  // 需要执行的 Contex，成功返回值为 0
  list<pair<Context*,int> > finisher_queue_rval;
                  // 需要执行的 Context，返回值为 int 类型的有效值
  ......
}
```

2.5　Throttle

类 Throttle 用来限制消费的资源数量（也常称为槽位“slot”），当请求的 slot 数量达

到 max 值时，请求就会被阻塞，直到有新的槽位释放出来，代码如下：

```
class Throttle {
  CephContext *cct;
  const std::string name;
  PerfCounters *logger;
  ceph::atomic_t count, max;
  // count: 当前占用的 slot 的数量
  // max:sloct 数量的最大值
  Mutex lock;          // 等待的锁
  list<Cond*> cond;   // 等待的条件变量
  ......
}
```

函数 get 用于获取数量为 c 个 slot，参数 c 默认为 1，参数 m 默认为 0，如果 m 不为默认的 0 值，就用 m 值重新设置 slot 的 max 值。如果成功获取数量为 c 个 slot，就返回 true，否则就阻塞等待。例如：

```
bool Throttle::get(int64_t c, int64_t m)
```

函数 get_or_fail 当获取不到数量为 c 个 slot 时，就直接返回 false，不阻塞等待：

```
bool Throttle::get_or_fail(int64_t c)
```

函数 put 用于释放数量为 c 个 slot 资源：

```
int64_t Throttle::put(int64_t c)
```

2.6 SafeTimer

类 SafeTimer 实现了定时器的功能，代码如下：

```
class SafeTimer
{
  CephContext *cct;
  Mutex& lock;
  Cond cond;
  bool safe_callbacks;        // 是否是 safe_callbacks

  SafeTimerThread *thread;   // 定时器执行线程

  std::multimap<utime_t, Context*> schedule;
                              // 目标时间和定时任务执行函数 Context
```

```
  std::map<Context*, std::multimap<utime_t, Context*>::iterator> events;
                                  //定时任务 <--> 定时任务在 shedule 中的位置映射
  bool stopping;                  //是否停止
}
```

添加定时任务的命令如下：

```
void SafeTimer::add_event_at(utime_t when, Context *callback)
```

取消定时任务的命令如下：

```
bool cancel_event(Context *callback);
```

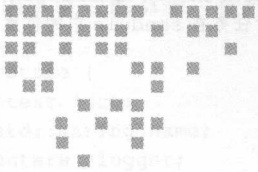
定时任务的执行如下：

```
void SafeTimer::timer_thread()
```

本函数一次检查 scheduler 中的任务是否到期，其循环检查任务是否到期执行。任务在 schedule 中是按照时间升序排列的。首先检查，如果第一任务没有到时间，后面的任务就不用检查了，直接终止循环。如果第一任务到了定时时间，就调用 callback 函数执行，如果是 safe_callbacks，就必须在获取 lock 的情况下执行 Callback 任务。

2.7　本章小结

本章介绍了 src/common 目录下的一些公共库中比较常见的类的实现。BufferList 在数据读写、序列化中使用比较多，它的各种不同成员函数的使用方法需要读者自己进一步了解。对于 ShardedThreadPool，本章只介绍了实现的原理，具体实现在不同的场景会有不同，需要读者面对具体的代码自己去分析。



Chapter 3 第 3 章

Ceph 网络通信

本章介绍 Ceph 网络通信模块，这是客户端和服务器通信的底层模块，用来在客户端和服务器之间接收和发送请求。其实现功能比较清晰，是一个相对较独立的模块，理解起来比较容易，所以首先介绍它。

3.1 Ceph 网络通信框架

一个分布式存储系统需要一个稳定的底层网络通信模块，用于各节点之间的互联互通。对于一个网络通信系统，要求如下：

- **高性能**。性能评价的两个指标：带宽和延迟。
- **稳定可靠**。数据不丢包，在网络中断时，实现重连等异常处理。

网络通信模块的实现在源代码 src/msg 的目录下，其首先定义了一个网络通信的框架，三个子目录里分别对应：Simple、Async、XIO 三种不同的实现方式。

Simple 是比较简单，目前比较稳定的实现，系统默认的用于生产环境的方式。它最大的特点是：每一个网络链接，都会创建两个的线程，一个专门用于接收，一个专门用于发送。这种模式实现比较简单，但是对于大规模的集群部署，大量的链接会产生大量的线

程，会消耗 CPU 资源，影响性能。

Async 模式使用了基于事件的 I/O 多路复用模式。这是目前网络通信中广泛采用的方式，但是在 Ceph 中，官方宣称这种方式还处于试验阶段，不够稳定，还不能用于生产环境。

XIO 方式使用了开源的网络通信库 accelio 来实现。这种方式需要依赖第三方的库 accelio 稳定性，需要对 accelio 的使用方式以及代码实现都比较熟悉。目前也处于试验阶段。特别注意的是，前两种方式只支持 TCP/IP 协议，XIO 可以支持 Infiniband 网络。

在 msg 目录下定义了网络通信的抽象框架，它完成了通信接口和具体实现的分离。在其下分别有 msg/simple 子目录、msg/Async 子目录、msg/xio 子目录，分别对应三种不同的实现。

3.1.1 Message

类 Message 是所有消息的基类，任何要发送的消息，都要继承该类，格式如图 3-1 所示。

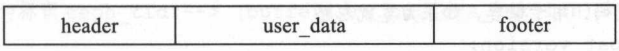


图 3-1　消息发送格式

消息的结构如下：header 是消息头，类似一个消息的信封（envelope），user_data 是用于要发送的实际数据，footer 是一个消息的结束标记，如下所示：

```
class Message : public RefCountedObject {
  ceph_msg_header  header;        // 消息头
  ceph_msg_footer  footer;        // 消息尾
  // 用户数据
  bufferlist       payload;       // "front" unaligned blob
  bufferlist       middle;        // "middle" unaligned blob
  bufferlist       data;          // data payload (page-alignment will be
preserved where possible)
  // 消息相关的时间戳
  utime_t recv_stamp;             // 开始接收数据的时间戳
  utime_t dispatch_stamp;         // dispatch 的时间戳
  utime_t throttle_stamp;         // 获取 throttle 的 slot 的时间戳
  utime_t recv_complete_stamp;    // 接收完成的时间戳
```

```
  ConnectionRef connection;       // 网络连接类
  uint32_t magic;                 // 消息的魔术字
  bi::list_member_hook<> dispatch_q;  //boost::intrusive 需要的字段
}
```

下面分别介绍其中的重要参数。

ceph_msg_header 为消息头，它定义了消息传输相关的元数据：

```
struct ceph_msg_header {
  __le64 seq;                   // 当前 session 内消息的唯一序号
  __le64 tid;                   // 消息的全局唯一的 id
  __le16 type;                  // 消息类型
  __le16 priority;              // 优先级
  __le16 version;               // 消息编码的版本
  __le32 front_len;             // payload 的长度
  __le32 middle_len;            // middle 的长度
  __le32 data_len;              // data 的长度

  __le16 data_off;              // 对象的数据偏移量

  struct ceph_entity_name src;// 消息源

  // 一些旧的代码，用于兼容，如果为零就忽略
  __le16 compat_version;
  __le16 reserved;
  __le32 crc;        // 消息头的 crc32c 校验信息
} __attribute__ ((packed));
```

ceph_msg_footer 为消息的尾部，附加了一些 crc 校验数据和消息结束标志：

```
struct ceph_msg_footer {
  __le32 front_crc, middle_crc, data_crc;
                    // 分别对应 crc 效验码
  __le64  sig;      // 消息的 64 位 signature
  __u8 flags;       // 结束标志
} __attribute__ ((packed));
```

消息带的数据分别保存在 payload、middle、data 这三个 bufferlist 中。payload 一般保存 Ceph 操作相关的元数据，middle 目前没有使用到，data 一般为读写的数据。

在源代码 src/messages 下定义了系统需要的相关消息，其都是 Message 类的子类。

3.1.2　Connection

类 Connection 对应端（port）对端的 socket 链接的封装。其最重要的接口是可以发送消息：

```
struct Connection : public RefCountedObject {
  mutable Mutex lock;              //锁保护 Connection 的所有字段
  Messenger *msgr;
  RefCountedObject *priv;          //链接的私有数据

  int peer_type;                   //链接的 peer 类型
  entity_addr_t peer_addr;         //peer 的地址
  utime_t last_keepalive, last_keepalive_ack;
                  //最后一次发送 keeplive 的时间和最后一次接收 keepalive 的 ACK 的时间

private:
  uint64_t features;               //一些 feature 的标志位
public:
  bool failed;                     //当值为 true 时，该链接为 lossy 链接已经失效了

  int rx_buffers_version;          //接收缓存区的版本

  map<ceph_tid_t,pair<bufferlist,int> > rx_buffers;  //接收缓冲区
        消息的标识 ceph_tid --> (buffer, rx_buffers_version) 的映射
}
```

其最重要的功能就是发送消息的接口：

```
virtual int send_message(Message *m) = 0;
```

3.1.3　Dispatcher

类 Dispatcher 是消息分发的接口，其分发消息的接口为：

```
virtual bool ms_dispatch(Message *m) = 0;
virtual void ms_fast_dispatch(Message *m);
```

Server 端注册该 Dispatcher 类用于把接收到的 Message 请求分发给具体处理的应用层。Client 端需要实现一个 Dispatcher 函数，用于处理收到的 ACK 应对消息。

3.1.4　Messenger

Messenger 是整个网络抽象模块，定义了网络模块的基本 API 接口。网络模块对外提

供的基本功能，就是能在节点之间发送和接受消息。

向一个节点发送消息的命令如下：

```
virtual int send_message(Message *m, const entity_inst_t& dest) = 0;
```

注册一个 Dispatcher 用来分发消息的命令如下：

```
void add_dispatcher_head(Dispatcher *d)
```

3.1.5 网络连接的策略

Policy 定义了 Messenger 处理 Connection 的一些策略：

```
struct Policy {
  bool lossy;       // 如果为 true，该当该连接出现错误时就删除
  bool server;      // 如果为 true，为服务端，都是被动连接
  bool standby;     // 如果为 true，该连接处于等待状态
  bool resetcheck; // 如果为 true，该连接出错后重连

  // 该 connection 相关的流控操作
  Throttle *throttler_bytes;
  Throttle *throttler_messages;

  // 本地端的一些 feature 标志
  uint64_t features_supported;
  // 远程端需要的一些 feature 标志
  uint64_t features_required;
}
```

3.1.6 网络模块的使用

通过下面最基本的服务器和客户端的实例程序，了解如何调用网络通信模块提供的接口来完成收发请求消息的功能。

1. Server 程序分析

Server 程序源代码在 test/simple_server.cc 里，这里只展示有关网络部分的核心流程。

1）调用 Messenger 的函数 create 创建一个 Messenger 的实例，配置选项 g_conf->ms_type 为配置的实现类型，目前有三种方式：simple、async、xio：

```
messenger = Messenger::create(g_ceph_context,
            g_conf->ms_type, entity_name_t::MON(-1),
            "simple_server",
            0 /* nonce */);
```

2）设置 Messenger 的属性：

```
messenger->set_magic(MSG_MAGIC_TRACE_CTR);
messenger->set_default_policy(
  Messenger::Policy::stateless_server(CEPH_FEATURES_ALL, 0));
```

3）对于 Server，需要 bind 服务端地址：

```
r = messenger->bind(bind_addr);
if (r < 0)
    goto out;
common_init_finish(g_ceph_context);
```

4）创建一个 Dispatcher，并添加到 Messenger：

```
dispatcher = new SimpleDispatcher(messenger);
messenger->add_dispatcher_head(dispatcher);
```

5）启动 Messenger：

```
messenger->start();
messenger->wait(); // 本函数必须等 start 完成才能调用
```

SimpleDispatcher 函数里实现了 ms_dispatch，用于把接收到的各种请求消息分发给相关的处理函数。

2. Client 程序分析

源代码在 test/simple_client.cc 里，这里只展示有关网络部分的核心流程。

1）调用 Messenger 的函数 create 创建一个 Messenger 的实例：

```
messenger = Messenger::create(g_ceph_context, g_conf->ms_type,
            entity_name_t::MON(-1),
            "client",
            getpid());
```

2）设置相关的策略：

```
messenger->set_magic(MSG_MAGIC_TRACE_CTR);
```

```
messenger->set_default_policy(Messenger::Policy::lossy_client(0, 0));
```

3）创建 Dispatcher 类并添加，用于接收消息：

```
dispatcher = new SimpleDispatcher(messenger);
messenger->add_dispatcher_head(dispatcher);
dispatcher->set_active();
```

4）启动消息：

```
r = messenger->start();
if (r < 0)
    goto out;
```

5）下面开始发送请求，先获取目标 Server 的链接：

```
conn = messenger->get_connection(dest_server);
```

6）通过 Connection 来发送请求消息。这里的消息发送方式都是异步发送，接收到请求消息的 ACK 应答消息后，将在 Dispatcher 的 ms_dispatch 或者 ms_fast_dispatch 处理函数里做相关的处理：

```
Message *m;
for (msg_ix = 0; msg_ix < n_msgs; ++msg_ix) {
  // 如果需要，这里要添加实际的数据
  if (! n_dsize) {
    m = new MPing();
  } else {
    m = new_simple_ping_with_data("simple_client", n_dsize);
  }
  conn->send_message(m);
}
```

综上所述，通过 Ceph 的网络框架发送消息比较简单。在 Server 端，只需要创建一个 Messenger 实例，设置相应的策略并绑定服务端口，然后就设置一个 Dispatcher 来处理接收到的请求。在 Client 端，只需要创建一个 Messenger 实例，设置相关的策略和 Dispatcher 用于处理返回的应答消息。通过获取对应 Server 的 connection 来发送消息即可。

3.2 Simple 实现

Simple 在 Ceph 里实现比较早，目前也比较稳定，是在生产环境中使用的网络通信模块。如其名字所示，实现相对比较简单。下面具体分析一下，Simple 如何实现 Ceph 网络

通信框架的各个模块。

3.2.1　SimpleMessager

类 SimpleMessager 实现了 Messager 接口。

```
class SimpleMessenger : public SimplePolicyMessenger {
  Accepter accepter;     //用于接受客户端的链接请求
  DispatchQueue dispatch_queue; //接收到的请求的消息分发队列
  bool did_bind;     //是否绑定

    __u32 global_seq;//生成全局的消息 seq
     ceph_spinlock_t global_seq_lock;// 用于保护 global_seq

  //地址→pipe 映射
  ceph::unordered_map<entity_addr_t, Pipe*> rank_pipe;
  //正在处理的 pipes
  set<Pipe*> accepting_pipes;
  //所有的 pipes
  set<Pipe*>      pipes;
  //准备释放的 pipes
  list<Pipe*>     pipe_reap_queue;

  //内部集群的协议版本
  int cluster_protocol;
}
```

3.2.2　Accepter

类 Accepter 用来在 Server 端监听端口，接收链接，它继承了 Thread 类，本身是一个线程，来不断地监听 Server 的端口：

```
class Accepter : public Thread {
  SimpleMessenger *msgr;
  bool done;
  int listen_sd;    //监听的端口
  uint64_t nonce;
  ......
}
```

3.2.3　DispatchQueue

DispatchQueue 类用于把接收到的请求保存在内部，通过其内部的线程，调用

SimpleMessenger 类注册的 Dispatch 类的处理函数来处理相应的消息：

```
class DispatchQueue {
  ......
  mutable Mutex lock;
  Cond cond;

  class QueueItem {
    int type;
    ConnectionRef con;
    MessageRef m;
    ......
  };

  PrioritizedQueue<QueueItem, uint64_t> mqueue;      // 接收消息的优先队列

  set<pair<double, Message*> > marrival;
  // 接收到的消息集合 pair 为 (recv_time, message)

  map<Message *, set<pair<double, Message*> >::iterator> marrival_map;
  // 消息→所在集合位置的映射
    ......
};
```

其内部的 mqueue 为优先级队列，用来保存消息，marrival 保存了接收到的消息。marrival_map 保存消息在集合中的位置。

函数 DispatchQueue::enqueue 用来把接收到的消息添加到消息队列中，函数 DispatchQueue::entry 为线程的处理函数，用于处理消息。

3.2.4 Pipe

类 Pipe 实现了 PipeConnection 的接口，它实现了两个端口之间的类似管道的功能。

对于每一个 pipe，内部都有一个 Reader 和一个 Writer 线程，分别用来处理这个 Pipe 有关的消息接收和请求的发送。线程 DelayedDelivery 用于故障注入测试：

```
class Pipe : public RefCountedObject {
  class Reader : public Thread {
  ......
  } reader_thread;
  // 接收线程，用于接收数据
  class Writer : public Thread {
```

```
    ......
    } writer_thread;
    //发送线程，用于发送数据
    SimpleMessenger *msgr;              // msgr 的指针
    uint64_t conn_id;                   //分配给 Pipe 自己唯一的 id

    char *recv_buf;                     //接收缓存区
    int recv_max_prefetch;              //接收缓冲区一次预取的最大值
    int recv_ofs;                       //接收的偏移量
    int recv_len;                       //接收的长度

    int sd;                             // pipe 对应的 socked fd

    struct iovec msgvec[IOV_MAX]; //发送消息的 iovec 结构

    int port;                           //链接端口
    int peer_type;                      //链接对方的类型
    entity_addr_t peer_addr;            //对方地址
    Messenger::Policy policy;           //策略

    Mutex pipe_lock;
    int state;                          //当前链接的状态
    atomic_t state_closed;              //如果非 0，那么状态为 STATE_CLOSED

    PipeConnectionRef connection_state;    //PipeConnection 的引用

    utime_t backoff;                    // backoff 的时间


    map<int, list<Message*> > out_q;   //准备发送的消息优先队列
    DispatchQueue *in_q;                //接收消息的 DispatchQueue
    list<Message*> sent;                //要发送的消息
    Cond cond;
    bool send_keepalive;
    bool send_keepalive_ack;
    utime_t keepalive_ack_stamp;
    bool halt_delivery;                 //如果 Pipe 队列消毁，停止增加

    __u32 connect_seq, peer_global_seq;
    uint64_t out_seq;                   //发送消息的序列号
    uint64_t in_seq, in_seq_acked;     //接收到消息序号和 ACK 的序号
}
```

3.2.5　消息的发送

1）当发送一个消息时，首先要通过 Messenger 类，获取对应的 Connection：

```
conn = messenger->get_connection(dest_server);
```

具体到 SimpleMessenger 的实现如下所示：

a）首先比较，如果 dest.addr 是 my_inst.addr，就直接返回 local_connection。

b）调用函数 _lookup_pipe 在已经存在的 Pipe 中查找。如果找到，就直接返回 pipeConnectionRef；否则调用函数 connect_rank 新创建一个 Pipe，并加入到 msgr 的 register_pipe 里。

2）当获得一个 Connection 之后，就可以调用 Connection 的发送函数来发送消息。

```
conn->send_message(m);
```

其最终调用了 SimpleMessenger::submit_message 函数：

a）如果 Pipe 不为空，并且状态不是 Pipe::STATE_CLOSED 状态，调用函数 pipe → _send 把发送的消息添加到 out_q 发送队列里，触发发送线程。

b）如果 Pipe 为空，就调用 connect_rank 创建 Pipe，并把消息添加到 out_q 发送队列中。

3）发送线程 writer 把消息发送出去。通过步骤 2，要发送的消息已经保存在相应 Pipe 的 out_q 队列里，并触发了发送线程。每个 Pipe 的 Writer 线程负责发送 out_q 的消息，其线程入口函数为 Pipe::writer，实现功能：

a）调用函数 _get_next_outgoing 从 out_q 中获取消息。

b）调用函数 write_message(header, footer, blist) 把消息的 header、footer、数据 blist 发送出去。

3.2.6 消息的接收

1）每个 Pipe 对应的线程 Reader 用于接收消息。入口函数为 Pipe::reader，其功能如下：

a）判断当前的 state，如果为 STATE_ACCEPTING，就调用函数 Pipe::accept 来接收连接，如果不是 STATE_CLOSED，并且不是 STATE_CONNECTING 状态，就接收消息。

b）先调用函数 tcp_read 来接收一个 tag。

c）根据 tag，来接收不同类型的消息如下所示：

- CEPH_MSGR_TAG_KEEPALIVE 消息。
- CEPH_MSGR_TAG_KEEPALIVE2，在 CEPH_MSGR_TAG_KEEPALIVE 的基础上，添加了时间。
- CEPH_MSGR_TAG_KEEPALIVE2_ACK。
- CEPH_MSGR_TAG_ACK。
- CEPH_MSGR_TAG_MSG，这里才是接收的消息。
- CEPH_MSGR_TAG_CLOSE。

d）调用函数 read_message 来接收消息，当本函数返回后，就完成了接收消息。

2）调用函数 in_q->fast_preprocess(m) 预处理消息。

3）调用函数 in_q->can_fast_dispatch(m)，如果可以进行 fast_dispatch，就 in_q->fast_dispatch(m) 处理。fast_dispatch 并不把消息加入到 mqueue 里，而是直接调用 msgr->ms_fast_dispatch 函数，并最终调用注册的 fast_dispatcher 函数处理。

4）如果不能 fast_dispatch，就调用函数 in_q->enqueue(m, m->get_priority(), conn_id) 把接收到的消息加入到 DispatchQueue 的 mqueue 队列里，由 DispatchQueue 的分发线程调用 ms_dispatch 处理。

ms_fast_dispath 和 ms_dispatch 两种处理的区别在于：ms_dispatch 是由 DispatchQueue 的线程处理的，它是一个单线程；ms_fast_dispatch 函数是由 Pipe 的接收线程直接调用处理的，因此性能比前者要好。

3.2.7　错误处理

网络模块复杂的功能是如何处理网络错误。无论是接收还是发送，会出现各种异常错误，包括返回异常错误码，接收数据的 magic 验证不一致，接收的数据的效验验证不一致，等等。错误的原因主要是由于网络本身的错误（物理链路等），或者字节跳变引起的。

目前错误处理的方法比较简单，处理流程如下：

1）关闭当前 socket 的连接。

2）重新建立一个 socket 连接。

3）重新发送没有接受到 ACK 应对的消息。

函数 Pipe::fault 用来处理错误：

1）调用 shutdown_socket 关闭 pipe 的 socket。

2）调用函数 requeue_sent 把没有收到 ACK 的消息重新加入发送队列，当发送队列有请求时，发送线程会不断地尝试重新连接。

3.3 本章小结

本章介绍了 Ceph 的网络通信模块的框架，及目前生产环境中使用的 Simple 实现。它对每个链接都会有一个发送线程和接收线程用来处理发送和接收。实现的难点还在于网络链接出现错误时的各种错误处理。

第 4 章 Chapter 4

CRUSH 数据分布算法

本章介绍 Ceph 的数据分布算法 CRUSH，它是一个相对比较独立的模块，和其他模块的耦合性比较少，功能比较清晰，比较容量理解。在客户端和服务器都有 CRUSH 的计算，了解它可以更好地理解后面的章节。

CRUSH 算法解决了 PG 的副本如何分布在集群的 OSD 上的问题。本章首先介绍 CRUSH 算法的原理，并给出相应的示列，然后进一步分析其实现的一些核心代码。

4.1 数据分布算法的挑战

存储系统的数据分布算法要解决数据如何分布到集群中的各个节点和磁盘上，其面临如下的挑战：

- **数据分布和负载的均衡**。首先是数据分布均衡，使数据能均匀地分别在各个节点和磁盘上。其次是负载均衡，使数据访问（读写等操作）的负载在各个节点和磁盘上的负载均衡。
- **灵活应对集群伸缩**。系统可以方便地增加或者删除存储设备（包括节点和设备失效的处理）。当增加或者删除存储设备后，能自动实现数据的均衡，并且迁移的数据

尽可能地少。

- **支持大规模集群**。为了支持大规模的存储集群，就要求数据分布算法维护的元数据相对较小，并且计算量不能太大。随着集群规模的增加，数据分布算法的开销比较小。

在分布式存储系统中，数据分布算法对于分布式存储系统至关重要。目前有两种基本实现方法，一种是基于集中式的元数据查询的方式，如 HDFS 的实现：文件的分布信息（layout 信息）是通过访问集中式元数据服务器获得；另一种是基于分布式算法以计算获得。例如一致性哈希算法（DHT）等。Ceph 的数据分布算法 CRUSH 就属于后者。

4.2 CRUSH 算法的原理

CRUSH 算法的全称为：Controlled、Scalable、Decentralized Placement of Replicated Data，可控的、可扩展的、分布式的副本数据放置算法。

由第 1 章中介绍过的 RADOS 对象寻址过程可知，CRUSH 算法解决 PG 如何映射到 OSD 列表中。其过程可以看成函数：

```
CRHUSH(X) → (OSDi, OSDj, OSDk)
```

输入参数：

- X 为要计算的 PG 的 pg_id。
- Hierachical Cluster Map 为 Ceph 集群的拓扑结构。
- Placement Rules 为选择策略。

输出一组可用的 OSD 列表。

下面将分别详细介绍 Hierarchical Cluster Map 的定义和组织方式。Placement rules 定义了副本选择的规则。最后介绍 Bucket 随机选择算法的实现。

4.2.1 层级化的 Cluster Map

层级化的 Cluster Map 定义了 OSD 集群具有层级关系的静态拓扑结构。OSD 的层级使得 CRUSH 算法在选择 OSD 时实现了机架感知（rack awareness）的能力，也就是通过规

则定义，使得副本可以分布在不同的机架、不同的机房中，提供数据的安全性。

层级化的 Cluster Map 的一些基本概念如下：

- Device：最基本的存储设备，也就是 OSD，一个 OSD 对应一个磁盘存储设备。
- bucket：设备的容器，可以递归的包含多个设备或者子类型的 bucket。bucket 的类型：bucket 可以有很多的类型，例如 host 就代表了一个节点，可以包含多个 device。Rack 就是机架，包含多个 host 等。在 Ceph 里默认的有 root、datacenter、room、row、rack、host 六个等级。用户也可以自己定义新的类型。每个 device 都设置了自己的权重，和自己的存储空间相关。bucket 的权重就是子 bucket（或者设备）的权重之和。

下列举例说明 bucket 的用法。

例 4-1　Cluster Map 的定义

```
host test1 {                          // 类型 host，名字为 test1
  id -2                               // bucket 的 id，一般为负值
  # weight 3.000                      // 权重，默认为子 item 的权重之和
  alg straw                           // bucket 随机选择的算法
  hash 0                              // bucket 随机选择的算法使用的 hash 函数，这里 0 代表使用
                                      hash 函数 jenkins1
  item osd.1 weight 1.000             // item1：osd.1 和权重值
  item osd.2 weight 1.000
  item osd.3 weight 1.000
}

host test2{
  id -3
  # weight 3.000
  alg straw
  hash 0
  item osd.3 weight 1.000
  item osd.4 weight 1.000
  item osd.5 weight 1.000
}

root default{                         // root 类型的 bucket，名字为 default
  id -1                               // id 号
  # weight 6.000
  alg straw                           // 随机选择的算法
  hash 0                              // rjenkins1
  item test1 weight 3.000
  item test2 weight 3.000
}
```

根据上面 Cluster Map 的语法定义，图 4-1 给出了比较直观的层级化的树型结构。

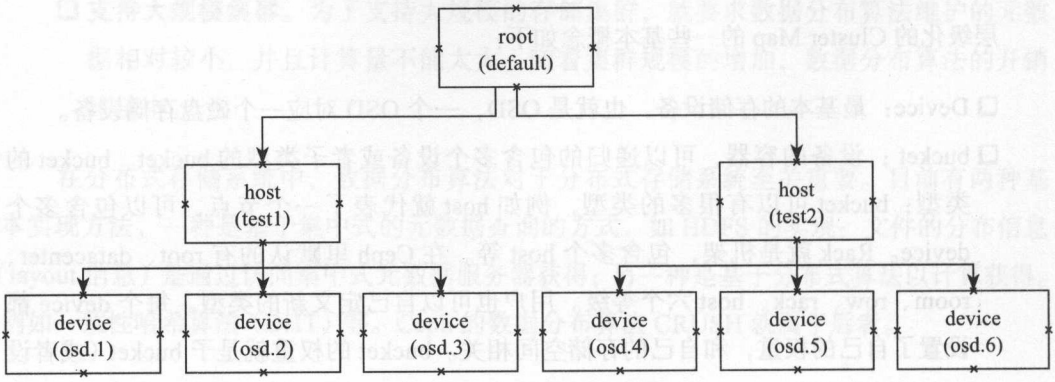


图 4-1 Cluster Map 示例图

在上面的 Cluster Map 定义中：

- 有一个 root 类型的 bucket，名字为 default。
- root 下面有两个 host 类型的 bucket，名字分别为 test1 和 test2，其下分别有三个 osd 设备，每个 device 的权重都为 1.000，说明它们的容量大小都相同。host 的权重为子设备之和为 3.000，它是自动计算的，不需要设置。
- Hash 设置了使用的 hash 函数，值 0 代表使用 rjenkins1 函数。
- alg 代表在该 bucket 里选择子 item 的算法。

4.2.2 Placement Rules

Cluster Map 反映了存储系统层级的物理拓扑结构。Placement Rules 决定了一个 PG 的对象副本如何选择的规则，通过这些可以自己设定规则，用户可以设定副本在集群中的分布。其定义格式如下：

```
tack(a)
choose
  choose firstn {num} type {bucket-type}
  chooseleaf firstn {num} type {bucket-type}、
    If {num} == 0, choose pool-num-replicas buckets (all available).
    If {num} > 0 && < pool-num-replicas, choose that many buckets.
    If {num} < 0, it means pool-num-replicas - {num}.
Emit
```

Placement Rules 的执行流程如下：

1）take 操作选择一个 bucket，一般是 root 类型的 bucket。

2）choose 操作有不同的选择方式，其输入都是上一步的输出：

a）choose firstn 深度优先选择出 num 个类型为 bucket-type 个的子 bucket。

b）chooseleaf 先选择出 num 个类型为 bucket-type 个子 bucket，然后递归到页节点，选择一个 OSD 设备：

- 如果 num 为 0，num 就为 pool 设置的副本数。
- 如果 num 大于 0，小于 pool 的副本数，那么就选择出 num 个。
- 如果 num 小于 0，就选择出 pool 的副本数减去 num 的绝对值。

3）emit 输出结果。

操作 chooseleaf firstn {num} type {bucket-type} 可以等同于两个操作：

a）choose firstn {num} type {bucket-type}

b）choose firstn 1 type osd

例 4-2 Placement Rules：三个副本分布在三个 Cabinet 中。

如图 4-2 所示的 Cluster Map：顶层是一个 root bucket，每个 root 下有四个 row 类型 bucket。每个 row 下面有 4 个 cabinet，每个 cabinet 下有若干个 OSD 设备（图中有 4 个 host，每个 host 有若干个 OSD 设备，但是在本 crush map 中并没有设置 host 这一级别的 bucket，而是直接把 4 个 host 上的所有 OSD 设备定义为一个 cabinet）：

```
rule replicated_ruleset {
  ruleset 0                          // ruleset 的编号 id
  type replicated                    // 类型 repliated 或者 erasure code
  min_size 1                         // 副本数最小值
  max_size 10                        // 副本数最大值
  step take root                     // 选择一个 root bucket，做下一步的输入
  step choose firstn  1 type row     // 选择一个 row，同一排
  step choose firstn  3 type cabinet // 选择三个 cabinet，三副本分别在不同的 cabinet
  step choose firstn  1 type osd     // 在上一步输出的三个 cabinet 中，分别选择一个 osd
  step emit
}
```

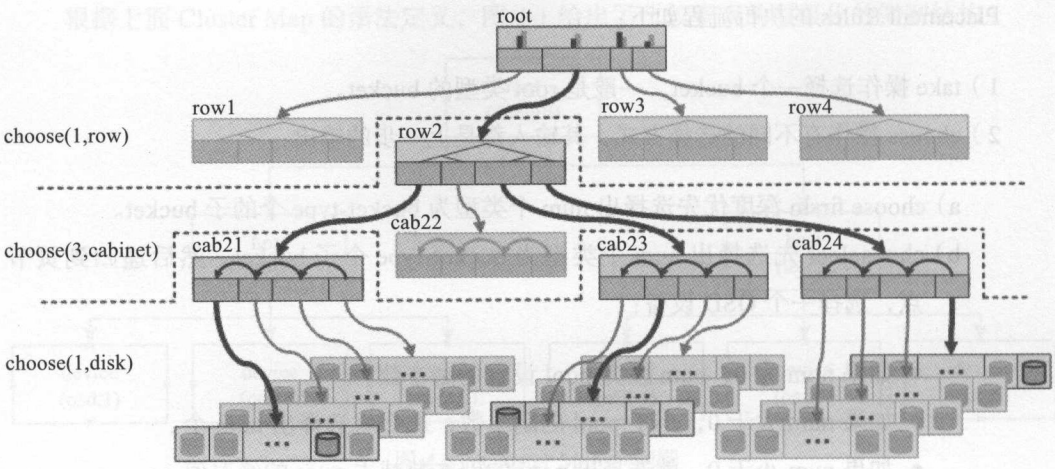


图 4-2 例 4-2 Cluster Map

根据上面的定义和图 4-2 的 Cluster Map 所示，选择算法的执行过程如下：

1）选中 root bucket 作为下一个步骤的输入。

2）从 root 类型的 bucket 中选择一个 row 类的子 bucket，其选择的算法在 root 的定义中设置，一般设置为 straw 算法。

3）从上一步的输出 row 中，选择三个 cabinet，其选择的算法在 row 中定义。

4）从上一步输出的三个 cabinet 中，分别选择一个 OSD，并输出。

根据本 rule sets，选择出三个 OSD 设备分布在一个 row 上的三个 cabinet 中。

例 4-3 Placement Rules：主副本分布在 SSD 上，其他副本分布在 HDD 上。

如图 4-3 所示的 Cluster Map：定义了两个 root 类型的 bucket，一个是名为 SSD 的 root 类型的 bucket，其 OSD 存储介质都是 SSD 盘。它包函两个 host，每个 host 上的设备都是 SSD 磁盘；另一个是名为 HDD 的 root 类型的 bucket，其 OSD 的存储介质都是 HDD 磁盘，它有两个 host，每个 host 上的设备都是 HDD 磁盘。

```
rule ssd-primary {
  ruleset 5
  type replicated
  min_size 5
  max_size 10
  step take ssd                          // 选择 ssd 这个 root bucket 为输入
```

```
    step chooseleaf firstn 1 type host    // 选择一个 host，并递归选择叶子节点 osd
    step emit                             // 输出结果
    step take hdd                         // 选择 hdd 这个 root bucket 为输入
    step chooseleaf firstn -1 type host
    // 选择总副本数减一个 host，并分别递归选择一个叶子节点 osd
    step emit                             // 输出结果
}
```

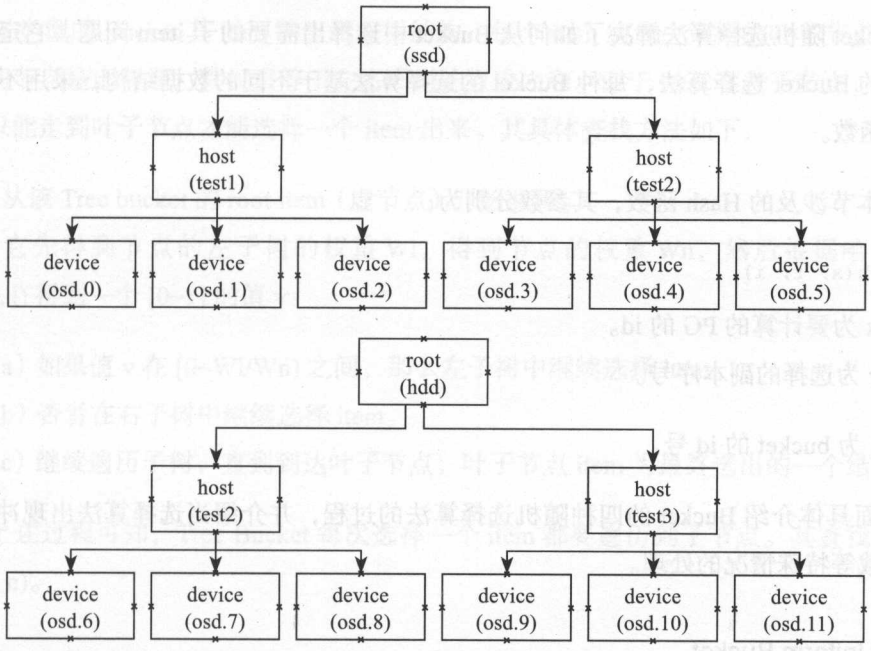


图 4-3　例 4-3 Cluster Map

根据图 4-3 所示的 Cluster Map，代码中的 rulesets 的执行过程如下：

1）首先 take 操作选择 ssd 为 root 类型的 bucket。

2）在 ssd 的 root 中先选择一个 host，然后以该 host 为输入，递归至叶子节点，选择一个 osd 设备。

3）输出选择的设备，也就是 ssd 设备。

4）选择 hdd 作为 root 的输入。

5）选择 2 个 host（副本数减一，默认 3 副本），并分别递归选择一个 OSD 设备，最终选择出两个 hdd 设备。

6）输出最终的结果。

最终输出 3 个设备，一个是 SSD 类型的磁盘，另外两个是 HDD 磁盘。通过上述规则，就可以把 PG 的主副本分布在 SSD 类型的 OSD 上，其他副本分布在 HDD 类型的磁盘上。

4.2.3 Bucket 随机选择算法

Bucket 随机选择算法解决了如何从 Bucket 中选择出需要的子 item 问题。它定义了四种不同的 Bucket 选择算法，每种 Bucket 的选择算法基于不同的数据结构，采用不同伪随机选择函数。

在本节涉及的 Hash 函数，其参数分别为；

```
hash(x, r, i)
```

- x 为要计算的 PG 的 id。
- r 为选择的副本序号。
- i 为 bucket 的 id 号。

下面具体介绍 Bucket 的四种随机选择算法的过程，并介绍当选择算法出现冲突、失效或过载等特殊情况的处理。

1. Uniform Bucket

Uniform 类型适用于每个 item，具有相同权重，且 item 很少添加和删除，也就是 item 的数量比较固定。它用了伪随机排列算法。

2. List Bucket

List 类型的 Bucket 中，其子 item 在内存中使用数据结构中的链表来保存，其所包含的 item 可以具有任意的权重。具体查找方法如下：

1）从 List Bucket 的表头 item 开始查找，它先得到表头 item 的权重 Wh，剩余链表中所有 item 的权重之和为 Ws。

2）根据本节提到的 hash(x, r, i) 函数得到一个 [0~1] 的值 v，假如这个值 v 在 [0~Wh/

Ws) 之中，则选择表头 item，并返回表头 item 的 id 值。

3）否者继续遍历剩余的链表，继续递归选择。

通过上述介绍可知，List 类型的 Bucekt 查找复杂度是 O(n)。

3. Tree Bucket

Tree 类型的 Bucket 其 item 的组织成树结构：每个 item 组成决策树的叶子节点。根节点和中间节点是虚节点，其权重等于左右子树的权重之和。由于 item 在叶子节点，所以每次选择只能走到叶子节点才能选择一个 item 出来。其具体查找方法如下：

1）从该 Tree bucket 的 root item（虚节点）开始遍历。

2）它先得到节点的左子树的权重 Wl，得到节点的权重 Wn，然后根据哈希函数 hash(x, r, i) 得到一个 [0~1] 的值 v：

a）如果值 v 在 [0~Wl/Wn) 之间，那么左子树中继续选择 item。

b）否者在右子树中继续选择 item。

c）继续遍历子树，直到到达叶子节点，叶子节点 item 为最终选出的一个结果。

由上述过程可知，Tree Bucket 每次选择一个 item 都要遍历到子节点。其查找复杂度是 O(log n)。

4. Straw Bucket

Straw 类的 Bucket 为默认的选择算法。该 Bucket 中的 item 选中概率是相同的，其实现如下：

1）函数 f(Wi) 为和 item 的权重 Wi 相关的函数，决定了每个 item 被选中的概率。

2）给每个 item 计算出一个长度，其计算公式为：

```
length = f(Wi)*hash(x, r, i)
```

length 值最大的 item 就是被选中的 item。

5. Bucket 选择算法的对比

如表 4-1 所示。

表 4-1 Bucket 选择算法对比

Bucket 选择算法	选择的速度	item 添加难易程度	item 删除难易程度
uniform	O(1)	poor	poor
list	O(n)	optimal	poor
tree	O(log n)	Good	Good
straw	O(n)	Better	Better
straw2	O(n)	optimal	optimal

算法 straw 比较容易应对 item 的添加和删除，为默认的 Bucket 选择算法。算法 straw2 是对算法 straw 的一些改进，可以减少数据的迁移数量。

6. 冲突、失效或者过载

当通过上述 Bucket 选择算法选出一个 OSD 后，有可能出现冲突（重复选择），该 OSD 已经失效了，或者过载（负载过重）的情况，就需要重新选择一次。

根据上述算法分析，选择时都依赖哈希函数：

```
hash(x, r, i)
```

其中，x 为 PG 的 id，r 为选择的副本数，i 为 Bucket 的 id 号。当选择出现上述情况需要重新选择。上述各种 Bucket 选择算法都依赖 hash 函数。当重新选择时，把参数 r 顺序增加即可通过上述 hash 函数重新计算一个新的 hash 值。

例 4-4 冲突选择过程

过程见表 4-2。

表 4-2 冲突选择过程示例

	r=0	r=1	r=2	r=3
pg 1.0	osd1	osd2	osd3	
pg 1.1	osd2	~~osd2~~	osd3	osd4

说明如下：

1）pg 1.0 根据副本 r 分别等于 0、1、2 来计算 hash（x，r，i），处理 OSD 列表 {osd1，osd2，osd3}。

2）pg 1.1 根据同样的方法：在 r 等于 0 时选择出了 osd2，在 r 等于 1 时又选择了 osd2，产生了冲突。这时就用 r 分别等于 2，3 来继续选择剩余的副本。最终 pg1.1 选择出的 OSD 列表为 {osd2，osd3，osd4}.

4.3　代码实现分析

在介绍了 CRUSH 算法的原理之后，下面就分析 CRUSH 算法实现的关键数据结构，并对算法具体实现函数进行分析。

4.3.1　相关的数据结构

CRUSH 算法相关的数据结构有 crush_map 结构、crush_bucket 结构和 crush_rule 结构，下面将详细介绍。

1. crush_map

结构 crush_map 定义了静态的所有 Cluster Map 的 bucket。bucket 为动态申请的二维数组，保存了所有的 bucket 结构。rules 定义了所有的 crush_rule 结构：

```
struct crush_map {
  struct crush_bucket **buckets;
  struct crush_rule **rules;
  ……
}
```

2. crush_bucket

结构 crush_bucket 用于保存 Bucket 相关的信息：

```
struct crush_bucket {
  __s32 id;          // bucket 的 id，一般为负值
  __u16 type;        // 类型，如果是 0，就是 OSD 设备
  __u8 alg;          // bucket 的选择算法
  __u8 hash;         // bucket 的 hash 函数
  __u32 weight;      // bucket 的权重
  __u32 size;        // bucket 下的 item 的数量
  __s32 *items;      // 子 bucket 在 crush_bucket 结构 buckets 数组的下标，这里特别要注
                        意的是，其子 item 的 cursh_bucket 结构体都统一保存在 crush map 结
```

```
                                    构中的 buckets 数组中，这里只保存其在数组中的下标

// 以下是随机排序选择算法的一些 Cache 的参数
  __u32 perm_x;      // 要选择的 x
  __u32 perm_n;      // 排列的总的元素
  __u32 *perm;       // 排列组合的结果
};
```

3. crush_rule

结构 crush_rule

```
struct crush_rule {
  __u32 len;                              //steps 的数组的长度
  struct crush_rule_mask mask;            //ruleset 相关的配置参数
  struct crush_rule_step steps[0]; // 操作步
};
struct crush_rule_mask {
  __u8 ruleset;                           //ruleset 的编号
  __u8 type;                              // 类型
  __u8 min_size;                          // 最新 size
  __u8 max_size;                          // 最大 size
};
struct crush_rule_step {
  __u32 op;                               //step 操作步的操作码
  __s32 arg1;                             // 如果是 take，参数就是选择的 bucket 的 id 号
                                          // 如果是 select，就是选择的数量
  __s32 arg2;                             // 如果是 select，是选择的类型
};
```

4.3.2 代码实现

代码 builder.c 和 builder.h 文件里主要实现了如何构造 crush_map 数据结构。Crush.c 和 Crush.h 文件里定义了 crush_map 相关的数据结构和 destroy 方法。文件 CrushCompiler.h 和 CrrushCompiler.cc 为 crush map 的词法和语法分析相关处理。类 CrushWarpper 是对 CRUSH 的所有核心实现进行的封装。CRUSH 算法的核心实现在 mapper.c 文件里。

1. crush_do_rule

函数 crush_do_rule 里完成了 CRUSH 算法的选择过程：

```
int crush_do_rule(const struct crush_map *map,      //crush map 结构
  int ruleno,                 //ruleset 的号
```

```
int x,                          // 输入，一般是 pg 的 id
int *result,                    // 输出 osd 列表
int result_max,                 // 输出 osd 列表的数量
  const __u32 *weight,          // 所有 osd 的权重，通过它来判断 osd 是否 out
int weight_max,                 // 所有 osd 的数量
int *scratch)
```

函数 cursh_do_rule 根据 step 的数量，循环调用相关的函数选择 bucket。如果是深度优先，就调用函数 crush_choose_firstn，如果是广度优先，就调用函数 crush_choose_indep 来选择。

2. crush_choose_firstn

函数调用 crush_bucket_choose 选择需要的副本数，并对选择出来的 OSD 做了相关的冲突检查，如果冲突或者失效或者过载，继续选择新的 OSD。

3. bucket 算法

函数 crush_bucket_choose 根据不同的类型 bucket，选择不同的算法来实现从 bucket 中选出 item，这里介绍最常用的 straw 算法：

```
static int bucket_straw_choose(struct crush_bucket_straw *bucket,
                               int x, int r)
```

函数 bucket_straw_choose 用于 straw 类型的 bucket 的选择，输入参数 x 为 pgid，r 为副本数，其具体实现如下：

1）对每个 item，计算 hash 值：

```
draw = crush_hash32_3(bucket->h.hash, x, bucket->h.items[i], r);
```

2）获取低 16 位，并乘以权重相关的修正值：

```
draw &= 0xffff;
draw *= bucket->straws[i];
```

3）选取 draw 值最大的 item 为选中的 item。

由上可知，这种算法类似抽签，是一种伪随机选择算法。

4.4 对 CRUSH 算法的评价

通过以上分析，可以了解到 CRUSH 算法实质是一种可分层确定性伪随机选择算法，它是 Ceph 分布式文件系统的一个亮点和创新。

优点如下：

- 输入元数据（cluster map、placement rule）较少，可以应对大规模集群。
- 可以应对集群的扩容和缩容。
- 采用以概率为基础的统计上的均衡，在大规模集群中可以实现数据均衡。

目前存在的缺点如下：

- 在小规模集群中，会有一定的数据不均衡现象。
- 增加新设备时，导致旧设备之间也有数据的迁移。

4.5 本章小结

本章介绍了 RADOS 的数据分布算法 CRUSH 的原理。Hierarchical Cluster Map 实质定义了存储集群的静态拓扑结构。Placement Rules 开放了数据副本的选择规则，可以由用户自己定义和编辑。Bucket 算法定义了从 bucket 选择一个 item 的算法。通过本章可以了解 CRUSH 算法的具体实现，它实质是一个可分层的伪随机分布选择算法，是 Ceph 的一个创新，但它并不完美，需要许多改进。

第 5 章 Chapter 5

Ceph 客户端

本章介绍 Ceph 的客户端实现。客户端是系统对外提供的功能接口，上层应用通过它来访问 Ceph 存储系统。本章首先介绍 Librados 和 Osdc 两个模块，通过它们可直接访问 RADOS 对象存储系统。其次介绍 Cls 扩展模块使用它们可方便地扩展现有的接口。最后介绍 Librbd 模块。由于 Librados 和 Librbd 的多数实现流程都比较类似，本章在介绍相关的数据结构后，只选取一些典型的操作流程介绍。

5.1 Librados

Librados 是 RADOS 对象存储系统访问的接口库，它提供了 pool 的创建、删除、对象的创建、删除、读写等基本操作接口。架构如图 5-1 所示。

在最上层是类 RadosClient，它是 Librados 的核心管理类，处理整个 RADOS 系统层面以及 pool 层面的管理。类 IoctxImpl 实现单个 pool 层的对象读写等操作。OSDC 模块实现了请求的封装和通过网络模块发送请求的逻辑，其核心类 Objecter 完成对象的地址计算、消息的发送等工作。

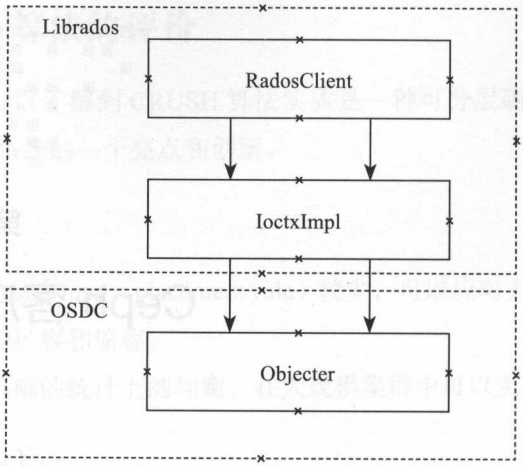


图 5-1 Librados 架构图

5.1.1 RadosClient

代码如下：

```
class librados::RadosClient : public Dispatcher
{
public:
  md_config_t *conf;                    // 配置文件
private:
  enum {
    DISCONNECTED,
    CONNECTING,
    CONNECTED,
  } state;                              // 和 Monitor 的网络连接状态

  MonClient monclient;                  //Monitor 客户端
  Messenger *messenger;                 // 网络消息接口

  uint64_t instance_id;                 //rados 客户端实例的 id
  Objecter *objecter;                   //objecter 对象指针

  Mutex lock;
  Cond cond;
  SafeTimer timer;                      // 定时器
  int refcnt;                           // 引用计算

  Finisher finisher;                    // 用于执行回调函数的 finisher 类
  ......
}
```

通过 RadosClient 的成员函数，可以了解 RadosClient 的实现功能如下：

1）网络连接。

Connect 函数是 RadosClient 的初始化函数，完成了许多的初始化工作：

a）调用函数 monclient.build_initial_monmap，从配置文件里检查是否有初始的 Monitor 的地址信息。

b）创建网络通信模块 messenger，并设置相关的 Policy 信息。

c）创建 objecter 对象并初始化。

d）调用 monclient.init() 函数初始化 monclient。

e）Timer 定时器初始化，Finisher 对象初始化。

2）pool 的同步和异步创建。

a）函数 pool_create 同步创建 pool。其实现过程为调用 Objecter::create_pool 函数，构造 PoolOp 操作，通过 Monitor 的客户端 monc 发送请求给 Monitor 创建一个 pool，并同步等待请求的返回。

b）函数 pool_create_async 异步创建。与同步方式的区别在于注册了回调函数，当创建成功后，执行回调函数通知完成。

3）pool 的同步和异步删除。

函数 delete_pool 完成删除，函数 delete_pool_async 异步删除。其过程和 pool 的创建过程相同，向 Monitor 发送删除的请求。

4）查找 pool 和列举 pool。

函数 lookup_pool 用于查找 pool，函数 pool_list 用于列出所有的 pool。pool 的相关的信息都保存在 OsdMap 信息当中。

5）获取 pool 和系统的信息。

函数 get_pool_stats 用于获取 pool 的统计信息，函数 get_fs_stats 用于获取系统的统计信息。

6）命令处理。

函数 mon_command 处理 Monitor 相关的命令，它调用函数 monclient.start_mon_command 把命令发送给 Monitor 处理。函数 osd_command 处理 OSD 相关的命令，它调用函数 objecter->osd_command 把命令发送给对应 OSD 处理。函数 pg_command 处理 PG 相关的命令，它调用函数 objecter->pg_command 把命令发送给该 PG 的主 OSD 来处理。

7）创建 IoCtxImpl 对象。

函数 create_ioctx 创建一个 pool 相关的上下文信息 IoCtxImpl 对象。

5.1.2 IoCtxImpl

类 IoCtxImpl 是 pool 相关的上下文信息，一个 pool 对应一个 IoCtxImpl 对象，可以在该 pool 里创建，删除对象，完成对象数据读写等各种操作，包括同步和异步的实现。其处理过程都比较简单，而且过程类似：

1）把请求封装成 ObjectOperation 类（该类定义在 osdc/Objecter.h 中）。

2）然后再添加 pool 的地址信息，封装成 Objecter::Op 对象。

3）调用函数 objecter->op_submit 发送给相应的 OSD，如果是同步操作，就等待操作完成。如果是异步操作，就不用等待，直接返回。当操作完成后，调用相应的回调函数通知。

5.2 OSDC

OSDC 是客户端比较底层的模块，其核心在于封装操作数据，计算对象的地址，发送请求和处理超时。

5.2.1 ObjectOperation

类 ObjectOperation 用于操作相关的参数统一封装在该类里，该类可以一次封装多个对象的操作：

```
struct ObjectOperation {
```

```
  vector<OSDOp> ops;                    // 多个操作
  int flags;                            // 操作的标志
  int priority;                         // 优先级

  vector<bufferlist*> out_bl;           // 每个操作对应的输出缓存区队列
  vector<Context*> out_handler;         // 每个操作对应的回调函数队列
  vector<int*> out_rval;                // 每个操作对应的操作结果队列
}
```

类 OSDOp 封装对象的一个操作。结构体 ceph_osd_op 封装一个操作的操作码和相关的输入和输出参数：

```
struct OSDOp {
  ceph_osd_op op;                       // 各种操作码和操作参数
  sobject_t soid;                       // 操作对象

  bufferlist indata, outdata;           // 输入和输出 bufferlist
  int32_t rval;                         // 操作结果
}
```

5.2.2　op_target

结构 op_target 封装了对象所在的 PG，以及 PG 对应的 OSD 列表等地址信息：

```
struct op_target_t {
  int flags;                            // 标志
  object_t base_oid;                    // 读取的对象
  object_locator_t base_oloc;           // 对象的 pool 信息
  object_t target_oid;                  // 最终读取的目标对象
  object_locator_t target_oloc;         // 最终目标对象的 pool 信息
  在这里由于 Cache tier 的存在，导致产生最终读取的目标和 pool 的不同。
}
```

5.2.3　Op

结构 Op 封装了完成一个操作的相关上下文信息，包括 target 地址信息、链接信息等：

```
struct Op : public RefCountedObject {
  OSDSession *session;      //OSD 相关的 Session 信息
  int incarnation;          // 引用次数

  op_target_t target;       // 地址信息

  vector<OSDOp> ops;        // 对应多个操作的封装

  snapid_t snapid;          // 快照的 id
```

```
  SnapContext snapc;                  // pool 层级的快照信息
  bufferlist *outbl;                  // 输出的 bufferlist
  vector<bufferlist*> out_bl;         // 每个操作对应的 bufferlist
  vector<Context*> out_handler;       // 每个操作对应的回调函数
  vector<int*> out_rval;              // 每个操作对应的输出结果
}
```

5.2.4 Striper

对象有分片（stripe）时，类 Striper 用于完成对象分片数据的计算。数据结构 ceph_file_layout 用来保存文件或者 image 的分片信息：

```
struct ceph_file_layout {                 // 文件 -> 对象的映射 *
  uint32_t fl_stripe_unit;                //stripe 的单位，必须是 page_size 的倍数
  uint32_t fl_stripe_count;               //stripe 跨越的对象数
  uint32_t fl_object_size;                // 对象的大小
  uint32_t fl_cas_hash;                   // 哈希值，当为 0 没有设置；当为 1=sha256 的哈希值

  /* pg -> disk layout 的映射 */
  uint32_t fl_object_stripe_unit;         // 没有用到

  /* 对象 -> pg layout 映射 */
  uint32_t fl_pg_preferred;               //PG 优先选择的主 OSD
  uint32_t fl_pg_pool;                    //pool 的 id
} __attribute__ ((packed));
```

对象 ObjectExtent 用来记录对象内的分片信息：

```
class ObjectExtent {
  public:
    object_t    oid;                  // 对象的 id
    uint64_t    objectno;             // 分片序号
    uint64_t    offset;               // 对象内的偏移
    uint64_t    length;               // 长度
    uint64_t    truncate_size;        // 对象 truncate 的操作的 size

    object_locator_t oloc;            // 对象位置信息，例如在哪个 pool 中，等等

    vector<pair<uint64_t,uint64_t> > buffer_extents; //Extents 在 buffer 中的偏移
                                                     和长度，有可能多个 extents
  }

void Striper::file_to_extents(
  CephContext *cct, const char *object_format,
  const ceph_file_layout *layout,   // 分片信息
  uint64_t offset, uint64_t len,    // 文件的偏移，长度
  uint64_t trunc_size,
  map<object_t,vector<ObjectExtent> >& object_extents, // 分布到每个对象的数据段
```

```
    uint64_t buffer_offset)     // 在 buffer 中的偏移量
}
```

函数 file_to_extents 完成了 file 到对象 stripe 后的映射。只有了解清楚了每个概念，计算方法都比较简单。下面举例说明。

例 5-1　file_to_extents 示例

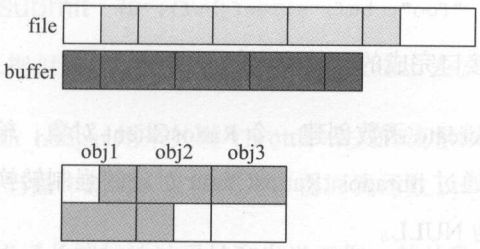


图 5-2　file_to_extents 示例

如图 5-2 所示，要计算的文件的 offset 为 2KB，length 为 16KB。文件的分片信息，stripe 为 4KB，stripe_count 为 3，object_size 为 8KB。对象 obj1 对应的 ObjectExtent 为：

```
object_extents[obj1] =
  {  oid  = "obj1",
     objectno = 0,
     offset = 2k,
     length = 6k,
     buffer_extents = { [0,2k], [6k,4k]}
  }
```

其中，oid 就是映射对象的 id，objectno 为 stirpe 对象的序号，offset 为映射的数据段在对象内的起始偏移，length 为对象内的长度。buffer_extents 为映射的数据在 buffer 内的偏移和长度。

5.2.5　ObjectCacher

类 ObjectCacher 提供了客户端的基于 LRU 算法对象数据缓存功能，其实比较简单，这里就不深入分析了。

5.3　客户写操作分析

以下代码是通过 Librados 库的接口写入数据到对象中的典型例程，对象的其他操作过

程都类似：

```
rados_t cluster;
rados_ioctx_t ioctx;
rados_create(&cluster, NULL);
rados_conf_read_file(cluster, NULL);
rados_connect(cluster);
rados_ioctx_create(cluster, pool_name.c_str(), &ioctx);
rados_write(ioctx, "foo", buf, sizeof(buf), 0)
```

上述代码是 C 语言接口完成的，其流程如下：

1）首先调用 rados_create 函数创建一个 RadosClient 对象，输出为类型 rados_t，它是一个 void 类型的指针，通过 librados::RadosClient 对象的强制转换产生。第二个参数 id 为一个标识符，一般传入为 NULL。

2）调用函数 rados_conf_read 来读取配置文件。第二个参数为配置文件的路径，如果是 NULL，就搜索默认的配置文件。

3）调用 rados_connect 函数，它调用了 RadosClient 的 connect 函数，做相关的初始化工作。

4）调用函数 rados_ioctx_create，它调用 RadosClient 的 create_ioctx 函数，创建 pool 相关的 IoCtxImpl 类，其输出为类型 rados_ioctx_t，它也是 void 类型的指针，由 IoCtxImpl 对象转换而来。

5）调用函数 rados_write 函数，向该 pool 的名为“foo”的对象写入数据。其调用 IoCtxImpl 类的 wrie 操作。

5.3.1 写操作消息封装

本函数完成具体的写操作：代码如下：

```
librados::IoCtxImpl::write(const object_t& oid, bufferlist& bl,
                           size_t len, uint64_t off)
```

其实现过程如下：

1）创建 ObjectOperation 对象，封装写操作的相关参数。

2）调用函数 operate 完成处理。

a）调用函数 objecter->prepare_mutate_op 把 ObjectOperation 类型的封装成 Op 类型，添加了 object_locator_t 相关的 pool 信息。

b）调用 objecter → op_submit 把消息发送出去。

c）等到操作完成。

5.3.2　发送数据 op_submit

函数 op_submit 用来把封装好的操作 Op 通过网络发送出去。

函数 _op_submit_with_budget 用来处理 Throttle 相关的流量限制。如果 osd_timeout 大于 0，就是设置定时器，当操作超时，就调用定时器回调函数 op_cancel 取消操作。

函数 _op_submit 完成了关键的地址寻址和发送工作，其处理过程如下：

1）调用函数 _calc_target 来计算对象的目标 OSD。

2）调用函数 _get_session 获取目标 OSD 的链接，如果返回值为 -EAGAIN，就升级为写锁，重新获取。

3）检查当前的状态标志，如果当前是 CEPH_OSDMAP_PAUSEWR 或者 OSD 空间满，就暂时不发送请求，否则调用函数 _prepare_osd_op 准备请求的消息，调用函数 _send_op 发送出去。

5.3.3　对象寻址 _calc_target

函数 _calc_target 用于完成对象到 osd 的寻址过程：

```
int Objecter::_calc_target(op_target_t *t, epoch_t *last_force_resend, bool
any_change)
```

其处理过程如下：

1）首先根据 t->base_oloc.pool 的 pool 信息，获取 pg_pool_t 对象。

2）检查如果强制重发，force_resend 设置为 true。

3）检查 cache tier，如果是读操作，并且有读缓存，就设置 target_oloc.pool 为该 pool 的 read_tier 值；如果是写操作，并且有写缓存，就设置 target_oloc.pool 为该 pool 的 write_tier 值。

4）调用函数 osdmap->object_locator_to_pg 获取目标对象所在的 PG。

5）调用函数 osdmap->pg_to_up_acting_osds，通过 CRUSH 算分，获取该 PG 对应的 OSD 列表。

6）如果是写操作，target 的 OSD 就设置为主 OSD；如果是读操作，如果设置了 CEPH_OSD_FLAG_BALANCE_READS 标志，就随机选择一个副本读取。如果设置了 CEPH_OSD_FLAG_LOCALIZE_READS 标志，就尽可能选择本地副本读取。

5.4 Cls

Cls 是 Ceph 的一个模块扩展，它允许用户自定义对象的操作接口和实现方法，为用户提供了一种比较方便的接口扩展方式。目前 rbd 和 lock 等模块都使用了这种机制。

5.4.1 模块以及方法的注册

类 ClassHandler 用来管理所有的扩展模块。函数 register_class 用来注册模块：

```
class ClassHandler{
  CephContext *cct;
  Mutex mutex;
  map<string, ClassData> classes;    // 所有注册的模块：模块名→模块元数据信息
  ……
}
```

类 ClassData 描述了一个模块的相关的元数据信息。它描述一个扩展模块的相关信息，包括模块名、模块相关的操作方法以及依赖的模块：

```
struct ClassData {
  enum Status {
    CLASS_UNKNOWN,          // 初始未知状态
    CLASS_MISSING,          // 缺失状态（动态链接库找不着）
    CLASS_MISSING_DEPS,     // 依赖的模块缺失
    CLASS_INITIALIZING,     // 正在初始化
    CLASS_OPEN,             // 已经初始化（动态链接库以及加载成功）
  } status;                 // 当前模块的加载状态

  string name;              // 模块的名字
  ClassHandler *handler;    // 管理模块的指针
  void *handle;
```

```
  map<string, ClassMethod> methods_map;   // 模块下所有注册的方法
  map<string, ClassFilter> filters_map;   // 模块下所有注册过滤方法
  set<ClassData *> dependencies;          // 本模块依赖的模块
set<ClassData *> missing_dependencies;    // 缺失的依赖模块
}
```

ClassMethod 定义一个模块具体的方法名，以及函数类型：

```
struct ClassMethod {
  struct ClassHandler::ClassData *cls;  // 所属模块的 ClassData 的指针
  string name;                          // 方法名
  int flags;                            // 方法相关的标志
  cls_method_call_t func;               //C 类型函数指针
  cls_method_cxx_call_t cxx_func;       //C++ 类型函数指针
}
```

在 src/objclass/class_api.c 里定义了一些辅助函数用来注册模块以及方法：

❑ 注册一个模块如下：

```
int cls_register(const char *name, cls_handle_t *handle);
```

❑ 注册一个模块的方法如下：

```
int cls_register_method(cls_handle_t hclass, const char *method,
                        int flags, cls_method_call_t class_call, cls_method_
                        handle_t *handle)
```

5.4.2　模块的方法执行

模块方法的执行在类 ReplicatedPG 的函数 do_osd_ops 里实现。执行方法对应的操作码为 CEPH_OSD_OP_CALL 值：

```
int ReplicatedPG::do_osd_ops(OpContext *ctx, vector<OSDOp>& ops){
  ......
  case CEPH_OSD_OP_CALL:
  // 加载相关的模块
  ClassHandler::ClassData *cls;
     result = osd->class_handler->open_class(cname, &cls);
     assert(result == 0);    // 函数 init_op_flags() 已经对结果做了验证
  // 根据方法名获取方法
  ClassHandler::ClassMethod *method = cls->get_method(mname.c_str());

  // 执行方法
  result = method->exec((cls_method_context_t)&ctx, indata, outdata);
```

```
.....
}
```

5.4.3 举例说明

以 Cls 下的 rbd 扩展接口来说明模块的定义、注册和调用过程：

1）rbd 的定义和注册。在 cls_rbd.cc 的函数里，注册了 rbd 模块，以及自定义的方法：

```
cls_register("rbd", &h_class);
cls_register_cxx_method(h_class, "create",
        CLS_METHOD_RD | CLS_METHOD_WR,
        create, &h_create);
.....
```

2）cls_rbd_client.h 和 cls_rbd_client.cc 里定义了客户端访问 rbd 函数，其调用 ioctx → exec 函数，封装成 CEPH_OSD_OP_CALL 类型的 ObjectOperation 操作，发送给相应的 osd 处理。

3）cls_rbd.cc 里定义了相应的函数在服务端的实现，其把输入参数从 bufferlist 中解析处理，调用相应的实现，把输出结果封装到输出 bufferlist 中。

```
int create(cls_method_context_t hctx, bufferlist *in,
           bufferlist *out)
{
  string object_prefix;
  uint64_t features, size;
  uint8_t order;

  try {
    bufferlist::iterator iter = in->begin();
    ::decode(size, iter);
    ::decode(order, iter);
    ::decode(features, iter);
    ::decode(object_prefix, iter);
  } catch (const buffer::error &err) {
    return -EINVAL;
   .....
  }
```

通过以上介绍，可以了解 Cls 的扩展模块：如何定义、注册一个新的模块，以及调用机制等。

5.5 Librbd

Librbd 模块实现了 RBD（rados block device）接口，其基于 Librados 实现了对 RBD 的基本操作。Librbd 的架构如图 5-3 所示。

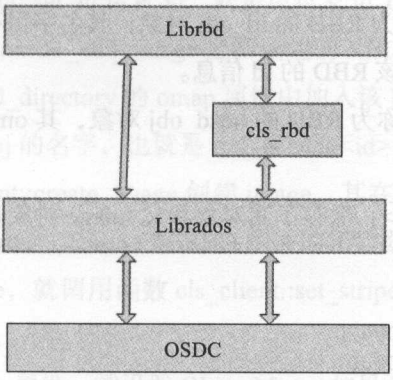


图 5-3　Librbd 的架构图

在最上层是 Librbd 层，模块 cls_rbd 是一个 Cls 扩展模块，实现了 RBD 的元数据相关的操作。RBD 的数据访问直接通过该 Librados 来访问。在最底层是 OSDC 层完成数据的发送。

5.5.1 RBD 的相关的对象

RBD 的相关对象如下所示。

- rbd_directory 对象：该对象在每个 pool 里都存在，用来保存该 pool 下所有的 RBD 的目录信息。当创建 RBD 块设备时，会检查如果 rbd_directory 对象不存在，就会创建该对象。该对象的 omap 属性里保存所有的 RBD 设备的名字和 id 信息。如表 5-1 所示。

表 5-1　rbd_directory 的 omap 的 kv 属性

Key 值	Value 值
"name_" + rbd_name1	"id_" + rbd_id1
"name_" + rbd_name2	"id_" + rbd_id2
……	……

omap 的属性里，保存所有 RBD 的设备名和 id 信息，key 值为“ name”和设备名的拼接，value 为“id_”和 RBD 的 id 的拼接。

❑ 对于每一个 RBD 设备，其创建如下对应的对象（最新的版本 v2）：

- rbd_id.<name>：称为 RBD 的 id_obj 对象，其名字保存了 RBD 的 name 信息，其对象的内容保存了该 RBD 的 id 信息。
- rbd_header.<id> ：称为 RBD 的 head_obj 对象，其 omap 保存了 RBD 相关的元数据信息。
- rbd_object_map.<id>：保存了其对象和父 image 对象映射信息。
- 数据对象（多个）。

```
rbd_data.<id>.00000000
rbd_data.<id>.00000001
......
```

5.5.2 RBD 元数据操作

结构 ImageCtx 用来处理一个 Image 的上下文相关信息。在 Internal.cc 和 Internal.h 定义了 RBD 的元数据相关的操作函数，其调用 Cls 下的 RBD 模块来完成 RBD 设备的创建、删除、快照等元数据操作。

下面通过分析 RBD 的创建过程来分析 RBD 的元数据操作过程，创建代码如下：

```
extern "C" int rbd_create4(rados_ioctx_t p, const char *name,
                           uint64_t size, rbd_image_options_t opts)
```

RBD 的创建根据传入参数的不同，有 4 个函数入口，下面主要研究 rbd_create4 函数，其实现过程如下：

1）调用函数 librbd::create 设置 RBD 相应的参数：

a）设置 RBD 的 format 格式。

b）设置 RBD 的 feature 信息。

c）设置 RBD 的分片信息 stripe_unit 和 stripe_count 参数。

d）设置 RBD 的 order 值，其决定了 RBD 对象 size 大小。

e）设置 bid 为 rados 的 instance_id，根据它来生成 RBD 的 id 值。

2）如果是版本 v2，就调用函数 create_v2 来继续创建：

a）获取 id_obj 的名字：rbd_id.<name>，调用函数 io_ctx.create 创建该对象。

b）和 bid 随机产生一个 id, 做为新的 image 的 id 值。

c）调用函数 cls_client::set_id 设置 id，就是在对象 id_obj 的内容中写入 id。

d）调用函数 cls_client::dir_add_image 把新创建的 RBD 加入 rbd_directory 目录中，也就是在对象 rbd_directory 的 omap 属性中加入该 RBD 的名字和 id 的键值对。

e）获取对象 head_obj 的名字，也就是 rbd_header.<id> 的格式。

f）调用函数 cls_client::create_image 创建 image，其在 head_obj 对象中的 omap 属性中设置 size、order、feature 等 RBD 相关的元数据信息。

g）如果对象有 stripe，就调用函数 cls_client::set_stripe_unit_count 设置 stripe 相关的参数。

h）如果有 ObjectMap 属性，就设置 ObjectMap，如果 RBD 有 mirror，就完成 rbd_journal 相关的设置。

通过 RBD 的创建过程，可以了解其元数据相关的操作，就是通过 Cls 的 RBD 模块设置相关的元数据信息。其他的元数据操作过程都类似。

5.5.3　RBD 数据操作

每个 ImageCtx 中都有一个对象 AioImageRequestWQ，它是一个工作队列，基于 ThreadPool::PointerWQ 来实现了异步请求的发送工作。

如图 5-4 所示：类 AioObjectRequest 是单个对象的异步请求。AioObjectRead 和 AbstractAioObjectWrite 分别为对象的异步读和异步写操作。对象的异步操作 truncate、write、trim、zero 等操作都继承 AbstractAioObjectWrite 类。

如图 5-5 所示：类 AioImageRequest 是所有 Image 操作的基类，AioImageRead 实现了 image 的异步读操作的逻辑。AbstractAioImageWrite 为异步写操作的抽象类。AioImageWrite，AioImageDiscard，AioImageFlush 继承了 AbstractAioImageWrite 类，分别完成了异步写，异步丢弃某一段数据（Discard），异步数据 Flush 等操作。

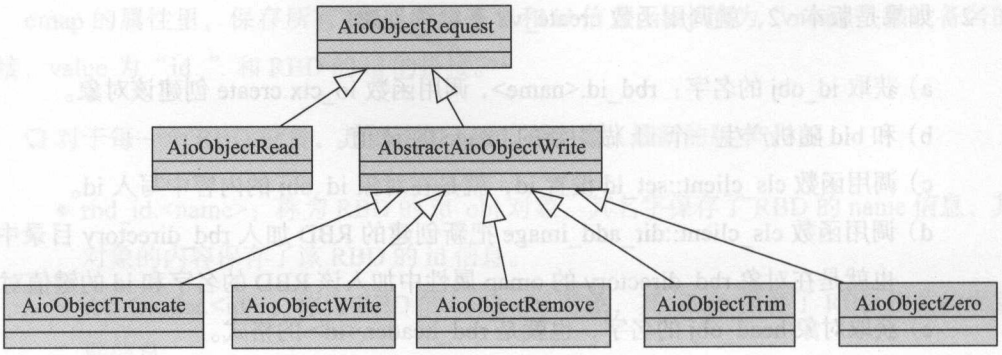


图 5-4 对象请求类图

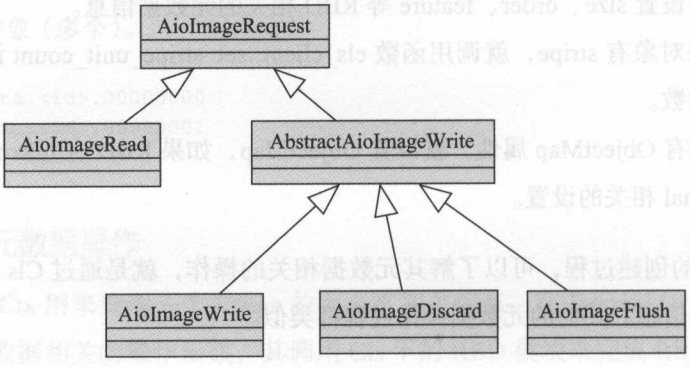


图 5-5 Image 请求类图

当一个 AioImageRequest 跨越多个对象时，每个对象就会产生一个 AioObjectRequest 请求，每个 AioObjectRequest 请求分别处理。

下面以 rbd_aio_write 为例，来介绍 RBD 的数据读写的流程：

```
extern "C" int rbd_aio_write(rbd_image_t image, uint64_t off, size_t len,
                             const char *buf, rbd_completion_t c)
```

其参数分别为：

- image：对象。
- off：image 内的偏移。
- len：写操作的长度。
- buf：写操作的数据 buf。
- 异步操作的回调函数。

处理流程如下：

1）调用 ictx->aio_work_queue->aio_write 函数来处理。

a）如果是非阻塞 IO，或者有 rbd mirror，或者有阻塞的 IO 请求，就调用函数 AioImageWrite 对象，加入到 AioImageRequestWQ 的工作队列里。

b）否则调用 AioImageRequest::aio_write，构建 AioImageWrite 对象，并调用 req.send() 发送。

2）加入 AioImageRequestWQ 队列的 AioImageWrite 请求，在线程池的处理函数 AioImageRequestWQ::process 里，同样调用 req.send() 函数来处理。

3）AioImageWrite 类的 send 函数继承自 AioImageRequest 的 send 函数，其直接调用 send_request 函数，AioImageWrite 的 send_request 函数继承来自 AbstractAioImageWrite 的 send_request 函数。

4）在 AbstractAioImageWrite 的 send_request 函数里，调用了 Striper::file_to_extents 函数，它计算该 image 的要写入的数据段映射到对象上的数据段。

5）在函数 send_object_requests 里，针对每一个对象，构建 AioObjectWrite 请求。如果有 rbd journal，就把请求加入到 aio_object_requests 里；否则就调用 AioObjectWrite 的 send 函数处理。

6）在 AbstractAioObjectWrite::send 函数里，调用 AbstractAioObjectWrite::send_pre 函数。该函数处理 object_map 相关的操作后；如果是写操，就调用 AbstractAioObjectWrite::send_write 函数。

7）在函数 AbstractAioObjectWrite::send_write 里，区分了两种操作：如果已经确认该对象不存在，并且有父 image，就 handle_write_guard 函数处理 clone 相关的 copyup 操作。否则就直接调用函数 AbstractAioObjectWrite::send_write_op 处理。

8）在函数 AbstractAioObjectWrite::send_write_op 里，调用 m_ictx->data_ctx.aio_operate 函数，通过该 rados 层的操作函数，完成对象数据的写入。

5.5.4　RBD 的快照和克隆

RBD 的快照基于 rados 的 snapshot 的机制实现。RBD 的快照在客户端的实现比较简单，其核心流程就是申请新的 snap_seq，把 snap_name 和 snap_id 记录在 RBD 的元数据

中。写操作的 copy-on-write 机制是依赖 rados 在 OSD 服务端实现的。

RBD 的 clone 操作在客户端实现了 RBD 的 copy-on-write 机制。当对 RBD 的 image 发起读操作或者写操作时，如果对象不存在，就需要检查：如果有有父 image，则读取父 image 所对应的对象数据。

1. RBD 的 snapshot 的创建

RBD 创建 snapshot 的核心逻辑在 SnapshotCreateRequest 里处理。其流程如下：

1）调用函数 SnapshotCreateRequest<I>::send_suspend_aio 来阻塞 RBD 的写操作。

2）调用函数 SnapshotCreateRequest<I>::send_allocate_snap_id 向 Monitor 申请一个新的 snap_seq 序号。

3）在函数 SnapshotCreateRequest<I>::send_create_snap() 里，调用 cls_client::snapshot_add 把 snap_name 和 snap_id 添加到 RBD 的 head_obj 的元数据中。

4）调用函数 ObjectMap::snapshot_add，构建 object_map::SnapshotCreateRequest，做 ObjectMap 相关的处理。

2. RBD 的 CopyUp 操作

当 RBD 在读写一个对象时，如果该对象不存在，并且有父 image，就需要 CopyUp 操作，读取父 image 所对应的对象数据。

其对应的实现在函数 void AbstractAioObjectWrite::send_write() 里，如果对象不存在，并且有 parent，就调用函数 handle_write_guard 处理：

```
if (!m_object_exist && has_parent()) {
      m_state = LIBRBD_AIO_WRITE_GUARD;
      handle_write_guard();
} else {
      send_write_op(true);
}
```

在函数 void AbstractAioObjectWrite::handle_write_guard 里，如果有 parent，就调用函数 send_copyup 函数：

```
if (has_parent) {
  send_copyup();
} else {
  // parent may have disappeared -- send original write again
  ldout(m_ictx->cct, 20) << "should_complete(" << this
    << "): parent overlap now 0" << dendl;
  send_write();
}
```

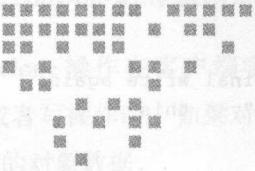
在函数 AbstractAioObjectWrite::send_copyup 里，构建 CopyupRequest 请求，来处理读取 parent image 对应的对象。

3. ObjectMap

在 RBD 里，新添加了一个特性，就是 ObjectMap，它添加了 RBD 的一个属性，用 Bit vector 的形式来记录一个对象是否存在，这样就可以极大地提供 clone 卷的读写性能。

5.6　本章小结

本章大致介绍了客户端的各个模块的功能以及核心典型操作的处理流程。由于 Librados 和 Librbd 的代码比较庞大，承载了所有功能的接口，故一些功能本章没有介绍到或者介绍得比较简略，但是客户端的代码有很大的类似性，通过了解典型流程，便不难理解其他代码。



Chapter 6 第 6 章

Ceph 的数据读写

本章介绍 Ceph 的服务端 OSD（书中简称 OSD 模块或者 OSD）的实现。其对应的源代码在 src/osd 目录下。OSD 模块是 Ceph 服务进程的核心实现，它实现了服务端的核心功能。本章先介绍 OSD 模块静态类图相关数据结构，再着重介绍服务端数据的写入和读取流程。

6.1 OSD 模块静态类图

OSD 模块的静态类图如图 6-1 所示。

OSD 模块的核心类及其之间的静态类图说明如下：

- 类 OSD 和类 OSDService 是核心类，处理一个 osd 节点层面的工作。在早期的版本中，OSD 和 OSDService 是一个类。由于 OSD 的类承载了太多的功能，后面的版本中引入 OSDService 类，分担一部原 OSD 类的功能。
- 类 PG 处理 PG 相关的状态维护以及实现 PG 层面的基本功能。其核心功能是用 boost 库的 statechart 状态机来实现的 PG 状态转换。
- 类 ReplicatedPG 继承了类 PG，在其基础上实现了 PG 内的数据读写以及数据恢复

相关的操作。

- 类 PGBackend 的主要功能是把数据以事务的形式同步到一个 PG 其他从 OSD 节点上。
  - PGBackend 的内部类 PGTransaction 就是同步的事务接口，其两个类型的实现分别对应 RPGTransaction 和 ECTransaction 两个子类。
  - PGBackend 两个子类 ReplicatedBackend 和 ECBackend 分别对应 PG 的两种类型的实现。
- 类 SnapMapper 额外保存对象和对象的快照信息，在对象的属性里保存了相关的快照信息。这里保存的快照信息为冗余信息，用于数据效验。

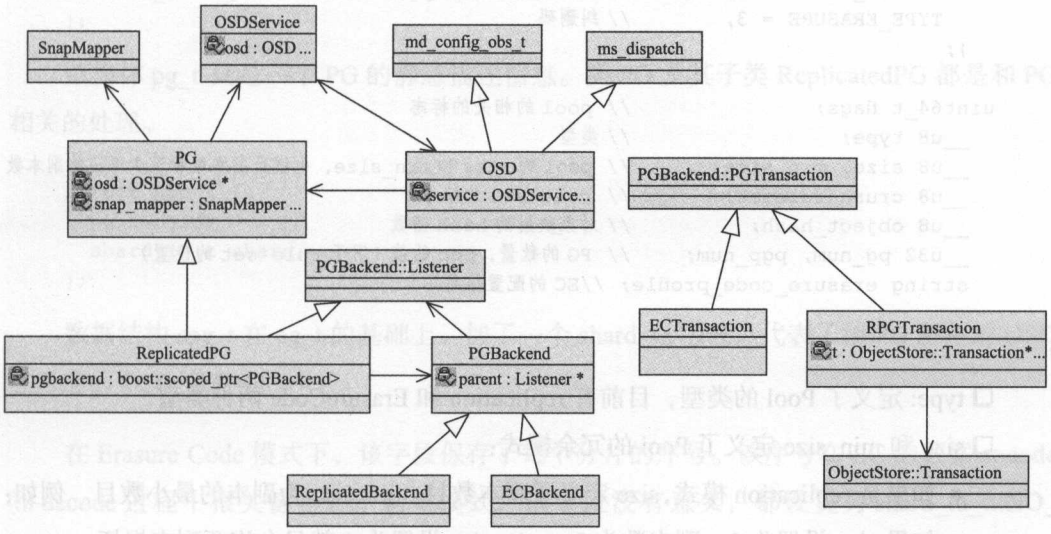


图 6-1　OSD 模块的静态类图

6.2　相关数据结构

下面将介绍 OSD 模块相关的一些核心的数据结构。从最高的逻辑层次为 pool 的概念，然后是 PG 的概念。其次是 OSDMap 记录了集群的所有的配置信息。数据结构 OSDOp 是一个操作上下文的封装。结构 object_info_t 保存了一个对象的元数据信息和访问信息。对象 ObjectState 是在 object_info_t 基础上添加了一些内存的状态信息。SnapSetContext 和 ObjectContext 分别保存了快照和对象的上下文相关的信息。Session 保

存了一个端到端的链接相关的上下文信息。

6.2.1 Pool

Pool 是整个集群层面定义的一个逻辑的存储池。对一个 Pool 可以设置相应的数据冗余类型，目前有副本和纠删码两种实现。数据结构 pg_pool_t 用于保存 Pool 的相关信息。Pool 的数据结构如下：

```
struct pg_pool_t {
  enum {
    TYPE_REPLICATED = 1,        // 副本
    //TYPE_RAID4 = 2,           // 从来没有实现的 raid4
    TYPE_ERASURE = 3,           // 纠删码
  };

uint64_t flags;                 // pool 的相关的标志
  __u8 type;                    // 类型
  __u8 size, min_size;          // pool 的 size 和 min_size，也就是副本数和至少保证的副本数
  __u8 crush_ruleset;           // rule set 的编号
  __u8 object_hash;             // 对象映射的 hash 函数
  __u32 pg_num, pgp_num;        // PG 的数量，pgp 的值（用于 rule set 的设置）
  string erasure_code_profile; //EC 的配置信息
.....
}
```

- ❑ type: 定义了 Pool 的类型，目前有 replication 和 ErasureCode 两种类型。
- ❑ size 和 min_size 定义了 Pool 的冗余模式：
  - 如果是 replication 模式，size 定义了副本数目，min_size 为副本的最小数目。例如：如果 size 设置为 3，副本数为 3，min_size 设置为 1 就只允许两副本损坏。
  - 如果是 Erasure Code（M+N），size 是总的分片数 M+N；min_size 是实际数据的分片数 M。
- ❑ crush_ruleset：Pool 对应的 crush 规则号。
- ❑ erasure_code_profile：EC 的配置方式。
- ❑ object_hash：通过对象名映射到 PG 的 hash 函数。
- ❑ pg_num：Pool 里 PG 的数量。

通过上面的介绍可以了解到，Pool 根据类型不同，定义了两种模式，分别保存了两种模式相关的参数。此外，在结构 pg_pool_t 里还定义了 Pool 级别的快照相关的数据结构、

Cache Tier 相关的数据结构，以及其他一些统计信息。在介绍快照（参见第 9 章）和 Cache Tier（参见第 13 章）时再详细介绍相关的字段。

6.2.2　PG

PG 可以认为是一组对象的集合，该集合里的对象有共同特征：副本都分布在相同的 OSD 列表中。PG 的数据结构如下：

```
struct pg_t {
  uint64_t m_pool;        //pg 所在的 pool
  uint32_t m_seed;        //pg 的序号
  int32_t  m_preferred;   //pg 优先选择的主 osd
};
```

结构体 pg_t 只是一个 PG 的静态描述信息。类 PG 及其子类 ReplicatedPG 都是和 PG 相关的处理。

```
struct spg_t {
  pg_t pgid;
  shard_id_t shard;
};
```

数据结构 spg_t 在 pg_t 的基础上，加了一个 shard_id 字段，代表了该 PG 所在的 OSD 在对应的 OSD 列表中的序号。

在 Erasure Code 模式下，该字段保存了每个分片的序号。该序号在 EC 的数据 encode 和 decode 过程中很关键；对于副本模式，该字段没有意义，都设置为 shard_id_t::NO_SHARD 值。

PG 的分裂　当一个 pool 里的 PG 数量不够时，系统允许通过命令增加 PG 的数量，就会产生 PG 的分裂，使得一个 PG 分裂为 2 的幂次方个 PG。PG 的分裂后，新的 PG 和其父 PG 的 OSD 列表是一致的，其数据的移动也是本地数据的启动，开销比较小。

6.2.3　OSDMap

类 OSDMap 定义了 Ceph 整个集群的全局信息。它由 Monitor 实现管理，并以全量或者增量的方式向整个集群扩散。每一个 epoch 对应的 OSDMap 都需要持久化保存在 meta 下对应对象的 omap 属性中。

下面介绍 OSDMap 核心成员，内部类 Incremental 以增量的形式保存了 OSDMap 新增的信息，其内部成员和 OSDMap 类似，这里就不介绍了。

```
class OSDMap {

  // 系统相关的信息
  uuid_d  fsid;                                    // 当前集群的 fsid 值
  epoch_t epoch;                                   // 当前集群的 epoch 值
  utime_t created, modified;                       // 创建修改的时间戳
  int32_t pool_max;                                // 最大的 pool 数量
  uint32_t flags;                                  // 一些标志信息

  //OSD 相关的信息
  int num_osd;                                     //OSD 的总数量
  int num_up_osd;                                  // 处于 up 状态的 OSD 的数量
  int num_in_osd;                                  // 处于 in 状态的 OSD 的数量
  int32_t max_osd;                                 //OSD 的最大数目
  vector<uint8_t> osd_state;                       //OSD 的状态
  ceph::shared_ptr<addrs_s> osd_addrs;             //OSD 的地址
  vector<__u32>   osd_weight;                      //OSD 的权重
  vector<osd_info_t> osd_info;                     //OSD 的基本信息
  ceph::shared_ptr< vector<uuid_d> > osd_uuid;  //OSD 对应的 uuid
  vector<osd_xinfo_t> osd_xinfo;                   //OSD 的一些扩展信息

  //PG 相关的信息
  ceph::shared_ptr< map<pg_t,vector<int32_t> > > pg_temp;
  // temp pg mapping (e.g. while we rebuild)
  ceph::shared_ptr< map<pg_t,int32_t > > primary_temp;
  // temp primary mapping (e.g. while we rebuild)
  ceph::shared_ptr< vector<__u32> > osd_primary_affinity;
  //< 16.16 fixed point, 0x10000 = baseline

  //pool 相关的信息
  map<int64_t,pg_pool_t> pools;                    //pool 的 id 到类 pg_pool_t 的映射
  map<int64_t,string> pool_name;                   //pool 的 id 到 pool 的名字的映射
  map<string,map<string,string> > erasure_code_profiles;  //pool 的 EC 相关的信息
  map<string,int64_t> name_pool;                   //pool 的名字到 pool 的 id 的映射

  //Crush 相关的信息
  ceph::shared_ptr<CrushWrapper> crush;     //CRUSH 算法
  ......
}
```

通过 OSDMap 数据成员的了解，可以看到，OSDMap 包含了四类信息：首先是集群的信息，其次是 pool 相关的信息，然后是临时 PG 相关的信息，最后就是所有 OSD 的状态信息。

6.2.4　OSDOp

类 MOSDOp 封装了一些基本操作相关的数据。

```
class MOSDOp : public Message {
  object_t oid;              // 操作的对象
  object_locator_t oloc;     // 对象的位置信息
  pg_t pgid;                 // 对象所在的 PG 的 id
  vector<OSDOp> ops;         // 针对 oid 的多个操作集合 private:

  // 快照相关
  snapid_t snapid;//snapid, 如果是 CEPH_NOSNAP, 就是 head 对象; 否则就是等于 snap_seq
  snapid_t snap_seq;
  // 如果是 head 对象 , 就是最新的快照序号
  // 如果是 snap 对象, 就是 snap 对应的 seq
  vector<snapid_t> snaps;   // 所有的 snap 列表
  uint64_t features;        // 一些 feature 的标志
  osd_reqid_t reqid;        // 请求的唯一 id 标识
```

MOSDOp 在其成员 ops 向量里分装了多个类型为 OSDOp 操作数据。MOSDOp 封装的操作都是关于对象 oid 相关的操作，一个 MOSDOp 只封装针对同一个对象 oid 的操作。但是对于 rados_clone_range 这样的操作，需要有一个目标对象 oid，还有一个源对象 oid，那么源对象的 oid 就保存在结构 OSDOp 里。

数据结构 OSDOp 封装了一个 OSD 操作需要的数据和元数据：

```
struct OSDOp {
  ceph_osd_op op;                  // 具体操作数据的封装
  sobject_t soid;
   //src oid, 并不是 op 操作的对象, 而是源操作对象
   // 例如 rados_clone_range 需要目标 obj 和源 obj
  bufferlist indata, outdata;  // 操作的输入输出的 data
  int32_t rval;                    // 操作返回值
  OSDOp() : rval(0) {
    memset(&op, 0, sizeof(ceph_osd_op));
  }
}
```

6.2.5　Object_info_t

结构 object_info_t 保存了一个对象的元数据信息和访问信息。其做为对象的一个属性，持久化保存在对象 xattr 中，对应的 key 为 OI_ATTR（”_”），value 就是 oject_info_t 的 encode 后的数据。

```
struct object_info_t {
  hobject_t soid;                          // 对应的对象
  eversion_t version, prior_version;       // 对象的当前版本，前一个版本
  version_t user_version;                  // 用户操作的版本
  osd_reqid_t last_reqid;                  // 最后请求的请求 id

  uint64_t size;                           // 对象的大小
  utime_t mtime;                           // 修改时间
  utime_t local_mtime;                     // 修改的本地时间

  typedef enum {
    FLAG_LOST     = 1<<0,
    FLAG_WHITEOUT = 1<<1,                  // object logically does not exist
    FLAG_DIRTY    = 1<<2,
    // object has been modified since last flushed or undirtied
    FLAG_OMAP     = 1 << 3,                // has (or may have) some/any omap data
    FLAG_DATA_DIGEST = 1 << 4,             // has data crc
    FLAG_OMAP_DIGEST = 1 << 5,             // has omap crc
    FLAG_CACHE_PIN = 1 << 6,               // pin the object in cache tier
    // ...
    FLAG_USES_TMAP = 1<<8,                 // deprecated; no longer used.
  } flag_t;

  flag_t flags;                            // 对象的一些标记
  ......
  vector<snapid_t> snaps;                  // clone 对象的快照信息

  uint64_t truncate_seq, truncate_size;
                                           //truncate 操作的序号和 size
  map<pair<uint64_t, entity_name_t>, watch_info_t> watchers;
   //watchers 记录了客户端监控信息，一旦对象的状态发送变化，需要通知客户端

  __u32 data_digest;                       //< data crc32c
  __u32 omap_digest;                       //< omap crc32c
  // 数据或者 omap 信息的 crc32 校验信息，可能有，也可能没有
}
```

6.2.6 ObjectState

对象 ObjectState 是在 object_info_t 基础上添加了一个字段 exists，用来标记对象是否存在。

```
struct ObjectState {
  object_info_t oi;
  bool exists;
  //the stored object exists (i.e., we will remember the object_info_t)
```

```
    ObjectState() : exists(false) {}
    ObjectState(const object_info_t &oi_, bool exists_)
      : oi(oi_), exists(exists_) {}
};
```

为什么要加一个额外的 bool 变量来标记呢？因为 object_info_t 可能是从缓存的 attrs[OI_ATTR] 中获取的，并不能确定对象是否存在。

6.2.7　SnapSetContext

SnapSetContext 保存了快照的相关信息，即 SnapSet 的上下文信息。关于 SnapSet 的内容，可以参考快照相关的介绍：

```
struct SnapSetContext {
  hobject_t oid;      // 对象
  int ref;            // 本结构的引用计数
  bool registered;    // 是否在 SnapSet Cache 中记录
  SnapSet snapset;    //SnapSet 对象快照相关的记录
  bool exists;        //snapset 是否存在

  SnapSetContext(const hobject_t& o) :
    oid(o), ref(0), registered(false), exists(true) {}
};
```

6.2.8　ObjectContext

ObjectContext 可以说是对象在内存中的一个管理类，保存了一个对象的上下文信息。

```
struct ObjectContext {
  ObjectState obs;                    // 主要是 object_info_t，描述了对象的状态信息
  SnapSetContext *ssc;                // 快照上下文信息，如果没有快照就为空
  Context *destructor_callback;       // 析构函数的
  private:
    Mutex lock;
  public:
  Cond cond;
  int unstable_writes, readers, writers_waiting, readers_waiting;
  // 正在写操作的数目，正在读操作的数目
  // 等待写操作的数目，等待读操作的数目

  // 如果该对象的写操作被阻塞去恢复另一个对象，设置这个属性
  ObjectContextRef blocked_by;        // 本对象被某个对象阻塞
  set<ObjectContextRef> blocking; // 本对象阻塞的对象集合
```

```
  // 任何在 obs.oi.watchers 中的 watchers 在 watchers 队列中或者在 unconnected_watchers 中
  map<pair<uint64_t, entity_name_t>, WatchRef> watchers;

  // 属性的缓存
  map<string, bufferlist> attr_cache;

  list<OpRequestRef> waiters;   // 等待状态变化的 waiters
  int count;                    // 读或写的数目

  struct RWState {
    enum State {
      RWNONE,
      RWREAD,
      RWWRITE,
      RWEXCL,
    };
    State state:4;                  // 读写的状态
    // 如果设置，获得锁后，重新执行 backfill 操作
    bool recovery_read_marker:1;
    // 如果设置，获得锁后重新加入 snaptrim 队列中
    bool snaptrimmer_write_marker:1;
  }
  ......
}
```

下面两个字段比较难理解，进行一些补充说明：

- blocked_by 记录了当前对象被其他对象阻塞，blocking 记录了本对象阻塞其他对象的情况。当一个对象的写操作依赖其他对象时，就会出现这些情况。这一般对应一个操作涉及多个对象，比如 copy 操作。把对象 obj1 上的部分数据拷贝到对象 obj2，如果源对象 obj1 处于 missing 状态，需要恢复，那么 obj2 对象就 block 了 obj1 对象。
- 内部类 RWState 通过定义了 4 种状态，实现了对对象的读写加锁。

6.2.9 Session

类 Session 是和 Connection 相关的一个类，用于保存 Connection 相关的上下文相关的信息。

```
struct Session : public RefCountedObject {
  EntityName entity_name;       //peer 实例的名字
  OSDCap caps;
```

```
  int64_t auid;
  ConnectionRef con;                    // 相关的 Connection
  WatchConState wstate;

  Mutex session_dispatch_lock;
  list<OpRequestRef> waiting_on_map;
    所有的 OpRequest 请求都先添加在这个队列里

  OSDMapRef osdmap;                     // Map as of which waiting_for_pg is current
  map<spg_t, list<OpRequestRef> > waiting_for_pg;
                                        // 当前需要更新 Osdmap 的 pg 和对应的请求

  Spinlock sent_epoch_lock;             // 通过消息向外通知的 epoch
  epoch_t last_sent_epoch;              // 发送对端的 epoch

  Spinlock received_map_lock;
  epoch_t received_map_epoch;           // 最新的 MOSDMap 消息接收到的 received_map_epoch

  Session(CephContext *cct) :
    RefCountedObject(cct),
    auid(-1), con(0),
    session_dispatch_lock("Session::session_dispatch_lock"),
    last_sent_epoch(0), received_map_epoch(0)
  {}
};
```

函数 update_waiting_for_pg 用于检查是否有最新的 osdmap：

1）如果该 PG 有分裂的 PG，就把分裂出的新的 PG 以及对应的 OpReqeust 加入到 session 的 waiting_for_pg 队列里。

2）如果该 PG 不分裂，就并把 PG 和 opRequest 加入到 waiting_for_pg 队列里。

6.3　读写操作的序列图

写操作序列图如图 6-2 所示。

写操作分为三个阶段：

- **阶段一**　从函数 ms_fast_dispatch 到函数 op_wq.queue 函数为止，其处理过程都在网络模块的回调函中处理，主要检查当前 OSD 的状态，以及 epoch 是否一致。
- **阶段二**　这个阶段在工作队列 op_wq 中的线程池里处理，在类 ReplicatedPG 里，

其完成对 PG 的状态、对象的状态的检查，并把请求封装成事务。

❑ **阶段三** 本阶段也是在工作队列 op_wq 中的线程池里处理，主要功能都在类 ReplicatedBackend 中实现。核心工作就是把封装好的事务通过网络分发到从副本上，最后调用本地 FileStore 的函数完成本地对象的数据写入。

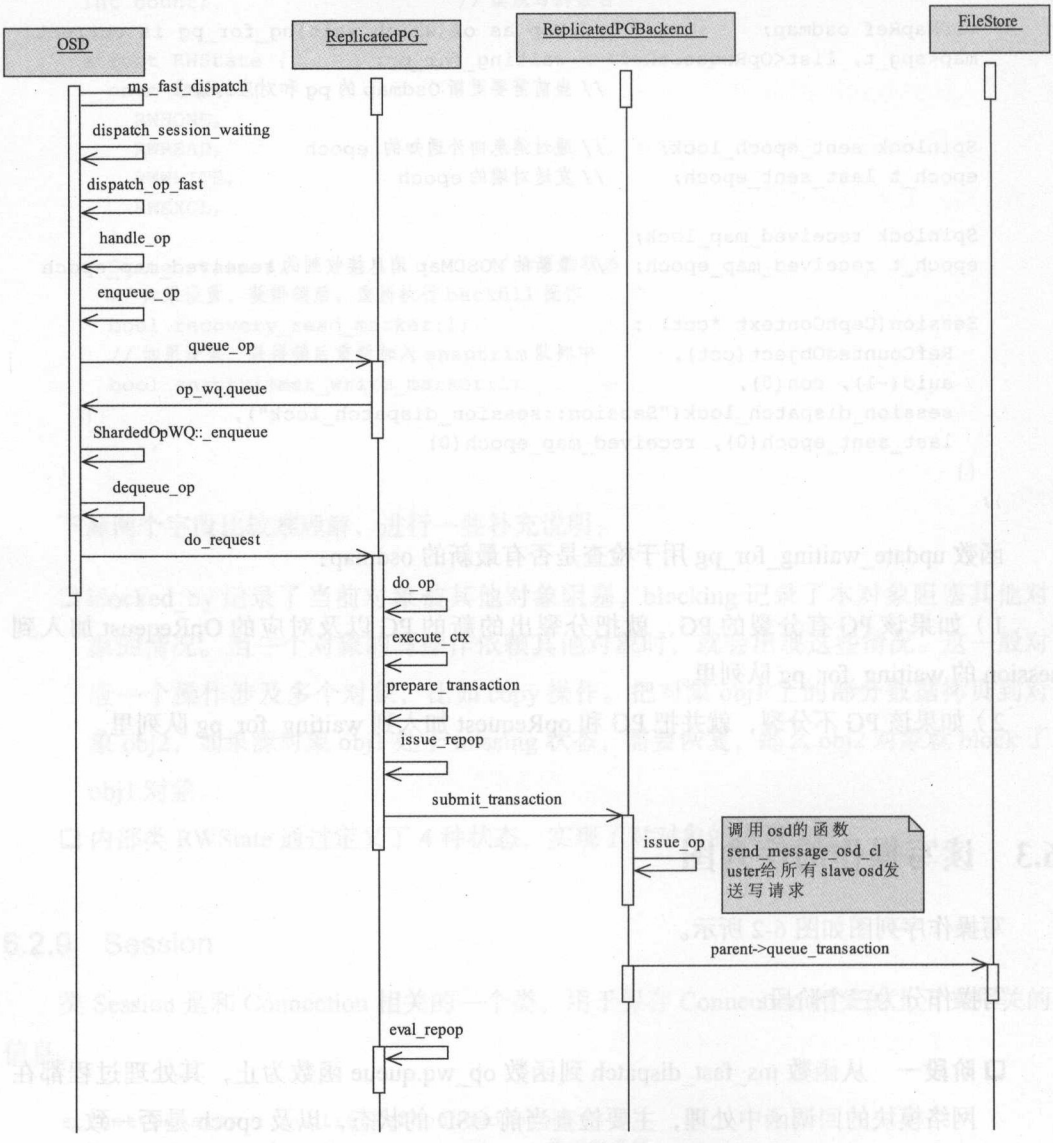


图 6-2 OSD 处理写操作的序列图

6.4　读写流程代码分析

在介绍了上述的数据结构和基本的流程之后，下面将从服务端接收到消息开始，分三个阶段具体分析读写的过程。

6.4.1　阶段 1：接收请求

读写请求都是从 OSD::ms_fast_dispatch 开始，它是接收读写消息 message 的入口。下面从这里开始读写操作的分析。本阶段所有的函数是被网络模块的接收线程调用，所以理论上应该尽可能的简单，处理完成后交给后面的 OSD 模块的 OpWQ 工作队列来处理。

1. ms_fast_dispatch

```
void OSD::ms_fast_dispatch(Message *m)
```

函数 ms_fast_dispatch 为 OSD 注册了网络模块的回调函数，其被网络的接收线程调用，具体实现过程如下：

1）首先检查 service，如果已经停止了，就直接返回。

2）调用函数 op_tracker.create_request 把 Message 消息转换为 OpRequest 类型，数据结构 OpRequest 包装了 Message，并添加了一些其他信息。

3）获取 nextmap（也就是最新的 osdmap）和 session，类 Session 保存了一个 Connection 的相关信息。

4）调用函数 update_waiting_for_pg 来更新 session 里保存的 OSDMap 信息。

5）把请求加入 waiting_on_map 的列表里。

6）调用函数 dispatch_session_waiting 处理，它循环调用函数 dispatch_op_fast 处理请求。

7）如果 session->waiting_on_map 不为空，说明该 session 里还有等待 osdmap 的请求，把该 session 加入到 session_waiting_for_map 队列里。

2. dispatch_op_fast

```
bool OSD::dispatch_op_fast(OpRequestRef& op, OSDMapRef& osdmap)
```

该函数检查 OSD 目前的 epoch 是否最新：

1）检查变量 is_stopping 如果 true，就直接返回 true 值。

2）调用函数 op_required_epoch(op)，从 OpRequest 中获取 msg 带的 epoch，进行比较，如果该值大于 OSD 最新的 epoch，则调用函数 osdmap_subscribe 更新 epoch，返回 flase 值。

3）否则，根据消息类型，调用相应的消息处理函数，本章只关注处理函数 handle_op 相关的流程。

3. handle_op

```
void OSD::handle_op(OpRequestRef& op, OSDMapRef& osdmap)
```

该函数处理 OSD 相关的操作，其处理流程如下：

1）首先调用 op_is_discardable，检查该 OP 是否可以丢弃。

2）构建 share_map 结构体，获取 client_session，从 client_session 获取 last_sent_epoch，调用函数 service.should_share_map 来设置 share_map.should_send 标志，该函数用于检查是否需要通知对方更新 epoch 值。这里和 dispatch_op_fast 的处理区别是，上次是更新自己，这里是通知对方更新。需要注意是，client 和 OSD 的 epoch 不一致，并不影响读写，只要 epoch 的变化不影响本次读写 PG 的 OSD list 变化。

3）从消息里获取 _pgid，从 _pg_id 里获取 pool。

4）调用函数 osdmap->raw_pg_to_pg，最终调用 pg_pool_t::raw_pg_to_pg 函数，对 PG 做了调整。

5）调用 osdmap->get_primary_shard 函数，获取该 PG 的主 OSD。

6）调用函数 get_pg_or_queue_for_pg，通过 pgid 获取 PG 类指针。如果获取成功，就调用函数 enqueue_op 处理请求。

7）如果 PG 类的指针没有获取成功，做一些错误检查：

a）seng_map 为空，client 需要重试。

b）客户端的 osdmap 里没有当前的 pool。

c）当前 OSD 的 osdmap 没有该 pool，或者当前 OSD 不是该 PG 的主 OSD。

总结，这个函数主要检查了消息的源端 epoch 是否需要 share，最主要的是获取读写请求相关的 PG 类后，下面就进入 PG 类的处理。

4. queue_op

```
void PG::queue_op(OpRequestRef& op)
```

该函数的实现如下：

1）加 map_lock 锁，该锁保护 waiting_for_map 列表，判断 waiting_for_map 列表不为空，就把当前 OP 加入该列表，直接返回。waiting_for_map 列表不为空，说明有操作在等待 osdmap 的更新，说明当前 osdmap 不信任，不能继续当前的处理。

2）函数 op_must_wait_for_map 判断当前的 epch 是否大于 OP 的 epoch，如果是，则必须加入 waiting_for_map 等待，等待更新 PG 当前的 epoch 值。

这里的 osdmap 的 epoch 的判断，是一个 PG 层的 epoch 的判断。和前面的判断不在一个层次，这里是需要等待的。

3）最终，把请求加入 OSD 的 op_wq 处理队列里。

总结，这个函数做在 PG 类里，做 PG 层面的相关检查，如果 ok，就加入 OSD 的 op_wq 工作队列里继续处理。

6.4.2　阶段 2：OSD 的 op_wq 处理

op_wq 是一个 ShardedWQ 类型的工作队列。以下操作都是在 op_wq 对应的线程池里调用做相应的处理。这里着重分析读写流程。

1. dequeue_op

```
void OSD::dequeue_op(PGRef pg, OpRequestRef op,ThreadPool::TPHandle &handle)
```

1）检查如果 op->send_map_update 为 true，也就是如果需要更新 osdmap，就调用函数 service.share_map 更新源端的 osdmap 信息。在函数 OSD::handle_op 里，只在 op->send_map_update 里设置了是否需要 share_map 的标记，在这里才真正去发消息实现 share 操作。

2）检查如果 pg 正在删除，就把本请求丢弃，直接返回。

3）调用函数 pg->do_request(op, handle) 处理请求。

总之，本函数主要实现了使请求源端更新 osdmap 的操作，接下来在 PG 里调 do_request 来处理。

2. do_request

本函数进入 ReplicatedPG 类来处理：

```
void ReplicatedPG::do_request( OpRequestRef& op, ThreadPool::TPHandle
&handle)
```

处理过程如下：

1）调用函数 can_discard_request 检查 op 是否可以直接丢弃掉。

2）检查变量 flushes_in_progress 如果还有 flush 操作，把 op 加入 waiting_for_peered 队列里，直接返回。

3）如果 PG 还没有 peered，调用函数 can_handle_while_inactive 检查 pgbackend 能否处理该请求，如果可以，就调用 pgbackend->handle_message 处理；否则加入 waiting_for_peered 队列，等待 PG 完成 peering 后再处理。

注意 PG 处于 inactive 状态，pgbackend 只能处理 MSG_OSD_PG_PULL 类型的消息。这种情况可能是：本 OSD 可能已经不在该 PG 的 acting osd 列表中，但是可能在上一阶段该 PG 的 OSD 列表中，所以 PG 可能含有有效的对象数据，这些对象数据可以被该 PG 当前的主 OSD 拉取以修复当前 PG 的数据。

4）此时 PG 处于 peered 并且 flushes_in_progress 为 0 的状态下，检查 pgbackend 能否处理该请求。pgbackend 可以处理数据恢复过程中的 PULL 和 PUSH 请求，以及主副本发的从副本的更新相关 SUBOP 类型的请求。

5）如果是 CEPH_MSG_OSD_OP，检查该 PG 的状态，如果处于非 active 或者 replay 状态，则把请求添加到 waiting_for_active 等待队列。

6）检查如果该 pool 是 cache pool，而该操作没有带 CEPH_FEATURE_OSD_CACHEPOOL 的 feature 标志，返回 EOPNOTSUPP 错误码。

7）根据消息的类型，调用相应的处理函数来处理。

本函数开始进入 ReplicatedPG 层面来处理，主要检查当前 PG 的状态是否正常，是否

可以处理请求。

3. do_op

函数 do_op 数比较复杂，处理读写请求的状态检查和一些上下文的准备工作。其中大量的关于快照的处理，本章在遇到快照处理时只简单介绍一下。

```
void ReplicatedPG::do_op(OpRequestRef& op)
```

具体处理过程如下：

1）调用函数 m->finish_decode，把消息带的数据从 bufferlist 中解析出相关的字段。

2）调用 osd->osd->init_op_flags 初始化 op->rmw_flags，函数 init_op_flags 根据 flag 来设置 rmw_flags 标志。

3）如果是读操作：

a）如果消息里带有 CEPH_OSD_FLAG_BALANCE_READS（平衡读）或者 CEPH_OSD_FLAG_LOCALIZE_READS（本地读）标志，表明主从副本都允许读。检查本 OSD 必须是该 PG 的 primary 或者 replica 之一。

b）如果没有上述标志，读操作只能读取主副本，本 OSD 必须是该 PG 的主 OSD。

4）如果里面含有 includes_pg_op 操作，调用 pg_op_must_wait 检查该操作是否需要等待，如果需要等待，加入 waiting_for_all_missing 队列；如果不需要等待，调用 do_pg_op 处理 PG 相关的操作。这里的 PG 操作，都是 CEPH_OSD_OP_PGLS 等类似的 PG 相关的操作，需要确保该 PG 上没有需要修复的对象，否则 ls 列出的对象就不准确。

5）调用函数 op_has_sufficient_caps 检查是否有相关的操作权限。

6）检查对象的名字是否超长。

7）检查操作的客是户端是否在黑名单（blacklist）中。

8）检查磁盘空间是否满。

9）检查如果是写操作，并且是 snap，返回 -EINVAL，快照不允许写操作。如果写操作带的数据大于 osd_max_write_size（如果设置了），直接返回 -OSD_WRITETOOBIG 错误。

可以看到，以上完成基本的与操作相关的参数检查。

10）构建要访问对象的 head 对象（head 对象和快照对象的概念可查看后面介绍快照

的章节）。

11）如果是顺序写，调用函数 scrubber.write_blocked_by_scrub 检查：如果 head 对象正在进行 scrub 操作，就加入 waiting_for_active 队列，等待 scrub 操作完成后继续本次请求的处理。

12）检查 head 对象是否处于缺失状态（missing）需要恢复，调用函数 wait_for_unreadable_object 把当前请求加入相应的队列里等待恢复完成。

13）如果是顺序写，检查 head 对象是否 is_degraded_or_backfilling_object，也就是正在恢复状态，需要调用 wait_for_degraded_object 加入相应的队列等待。

14）检查 head 对象的特殊情况：

a）检查队列 objects_blocked_on_degraded_snap 里如果保存的 head 对象，就需要等待。该队列里保存的 head 对象在 rollback 到某个版本的快照时，该版本的 snap 对象处于缺失状态，必须等待该 snap 对象恢复，从而完成 rollback 操作。因此该队列的 head 对象目前处于缺失状态。

b）队列 objects_blocked_on_snap_promotion 里的对象表示 head 对象 rollback 到某个版本的快照时，该版本的快照对象在 Cache pool 层没有，需要到 Data pool 层获取。

如果 head 对象在上述的两个队列中，head 对象都不能执行写操作，需要等待获取快照对象，完成 rollback 后才能写入。

可知，以上 10 ~ 14 步骤是构建并检查 head 对象的状态是否正常。

15）如果是顺序写操作，检查该对象是否在 objects_blocked_on_cache_full 队列中，该队列中的对象因 Cache pool 层空间满而阻塞写操作。

当 head 对象被删除时，系统自动创建一个 snapdir 对象用来保存快照相关的信息。head 对象和 snapdir 对象只能有一个存在，其都可以用来保存快照相关的信息。

16）检查该对象的 snapdir 对象（如果存在）是否处于 missing 状态。

17）检查 snapdir 对象是否可读，如果不能读，就调用函数 wait_for_unreadable_object 等待。

18）如果是写操作，调用函数 is_degraded_or_backfilling_object 检查 snapdir 对象是否缺失。

19）检查如果是 CEPH_SNAPDIR 类型的操作，只能是读操作。Snapdir 对象只能读取。

20）检查是否是客户端 replay 操作。

21）构建对象 oid，这才是实际要操作的对象，可能是 snap 对象也可能是 head 对象。

22）调用函数检查 maybe_await_blocked_snapset 是否被 block，检查该对象缓存的 ObjectContext 如果设置为 blocked 状态，该 object 有可能正在 flush，或者 copy（由于 Cache Tier），暂时不能写，需要等待。

23）调用函数 find_object_context 获取 object_context，如果获取成功，需要检查 oid 的状态。

24）如果 hit_set 不为空，就需要设置 hit_set. hit_set 和 agent_state 都是 Cache tier 的机制，hit_set 记录 cachepool 中对象是否命中，暂时不深入分析。

25）如果 agent_state 不为空，就调用函数 agent_choose_mode 设置 agent 的状态，调用函数 maybe_handle_cache 来处理，如果可以处理，就返回。

26）获取 object_locator，验证是否和 msg 里的相同。

27）检查该对象是否被阻塞。

28）获取 src_obc，也就是 src_oid 对应的 ObjectContext：同样的方法，对 src_oid 做各种状态检查，然后调用 find_object_context 函数获取 ObjectContext。

29）如果是操作对象 snapdir 对象，相关的操作就需要所有的 clone 对象，获取 clone 对象的 objectContext。对每一个 clone 对象，构建 objectContext，并把它加入的 src_obs 中。

30）创建 opContext。

31）调用 execute_ctx(ctx)。

总之，do_op 主要检查相关对象的（head 对象、snapdir 对齐、src 对象等）状态是否正常，并获取 ObjectContext、OpContext 相关的上下文信息。

4. get_object_context

本函数获取一个对象的 ObjectContext 信息。

```
ObjectContextRef ReplicatedPG::get_object_context(
                const hobject_t& soid,  //soid 要获取的对象
```

```
                bool can_create, // 是否允许创建新的 ObjectContext
                  map<string, bufferlist> *attrs) //attrs 对象的属性
```

关键是从属性 OI_ATTR 中获取 object_info_t 信息。具体过程如下：

1）首先从 LRU 缓存 object_contexts 的中获取该对象的 ObjectContext，如果获取成功，就直接返回结果。

2）如果从 LRU cache 里没有查找到：

a）如果参数 attrs 值不为空，就从 attrs 里获取 OI_ATTR 的属性值。

b）否则调用函数 pgbackend->objects_get_attr 获取该对象的 OI_ATTR 属性值。如果获取失败，并且不允许创建，就直接返回 ObjectContextRef() 的空值。

3）如果成功获取 OI_ATTR 属性值，就从该属性值中 decode 后获取 object_info_t 的值。

4）调用 get_snapset_context 获取 SnapSetContext。

5）调用相关函数设置 obc 相关的参数，并返回 obc。

5. get_snapset_context

本函数获取对象的 snapset_context 结构 ，其过程和函数 get_object_context 类似。具体实现如下：

1）首先从 LRU 缓存 snapset_contexts 获取该对象的 snapset_context，如果成功，直接返回结果。

2）如果不存在，并且 can_create，就调用 pgbackend->objects_get_attr 函数获取 SS_ATTR 属性。只有 head 对象或者 snapdir 对象保存有 SS_ATTR 属性，如果 head 对象不存在，就获取 snapdir 对象的 SS_ATTR 属性值，根据获得的值，decode 后获的 SnapsetContext 结构。

6. find_object_context

本函数查找对象的 object_context，这里需要理解 snapshot 相关的知识。根据 snap_seq 正确与否获取相应的 clone 对象，然后获取相应的 object_context。

```
int ReplicatedPG::find_object_context(const hobject_t& oid,     // 要查找的对象
  ObjectContextRef *pobc,       // 输出对象的 ObjectContext
```

```
bool can_create,                    //是否需要创建
bool map_snapid_to_clone,           //映射 snapid 到 clone 对象
hobject_t *pmissing                 //如果对象不存在，返回缺失的对象
)
```

参数 map_snapid_to_clone 指该 snap 是否可以直接对应一个 clone 对象，也就是 snap 对象的 snap_id 在 SnapSet 的 clones 列表中。

1）如果是 head 对象，就调用函数 get_object_context 获取 head 对象的 ObjectContext，如果失败，设置 head 对象为 pmissing 对象，返回 -ENOENT；如果成功，返回 0。

2）如果是 snapdir 对象，先获取 head 对象的 ObjectContext，如果失败，继续获取 snapdir 对象的 ObjectContext，如果失败，返回返回 -ENOENT；如果成功，返回 0。

3）如果非 map_snapid_to_clone 并且该 snap 已经标记删除了，就直接返回 -ENOENT，pmissing 为空，意味着该对象确实不存在。

4）调用函数 get_snapset_context 来获取 SnapSetContext，如果不存在，设置 pmissing 为 head 对象，返回 -ENOENT。

5）如果是 map_snapid_to_clone：

a）如果 oid.snap 大于 ssc->snapset.seq，说明该 snap 是最新做的快照，osd 端还没有完成相关的信息更新，直接返回 head 对象 object_context，如果 head 对象存在，就返回 0，否则返回 -ENOENT。

b）否则，直接检查 SnapSet 的 clones 列表，如果没有，就直接返回 -ENOENT。

c）如果找到，检查对象如果处于 missing，pmissing 就设置为该 clone 对象，返回 -EAGAIN。如果没有，就获取该 clone 对象的 object_context。

6）如果不是 map_snapid_to_clone，就不能从 snap_id 直接获取 clone 对象，需要根据 snaps 和 clones 列表，计算 snap_id 对应的 clone 对象：

a）如果 oid.snap > ssc->snapset.seq，获取 head 对象的 ObjectContext。

b）计算 oid.snap 首次大于 ssc->snapset.clones 列表中的 clone 对象，就是 oid 对应的 clone 对象。

c）检查该 clone 对象如果 missing，设置 pmissing 为该 clone 对象，返回 -EAGAIN。

d）获取该 clone 对象的 ObjectContext。

e）最后检查该 clone 对象是如果 first 和 last 之间，这是合理的情况，返回 0；否

则就是异常情况，返回 -ENOENT。

本函数是获取实际对象的 ObjectContext，如果不是 head 对象，就需要获取快照对象实际对应的 clone 对象的 ObjectContext。

7. execute_ctx

在 do_op 函数里，做了大量的对象状态的检查和上下文相关信息的获取，本函数开始执行相关的操作。

```
void ReplicatedPG::execute_ctx(OpContext *ctx)
```

处理过程如下：

1）首先在 OpContext 中创建一个新的事务，该事务为 pgbackend 定义的事务。

```
ctx->op_t = pgbackend->get_transaction()
```

2）如果是写操作，更新 ctx->snapc 值。ctx->snapc 值保存了该操作的客户端附带的快照相关信息：

a）如果是给整个 pool 的快照操作，就设置 ctx->snapc 等于 pool.snapc 的值。

b）如果是用户特定快照（目前只有 rbd 实现），ctx->snapc 值就设置为消息带的相关信息：

```
ctx->snapc.seq = m->get_snap_seq();
ctx->snapc.snaps = m->get_snaps();
```

c）如果设置了 CEPH_OSD_FLAG_ORDERSNAP 标志，客户端的 snap_seq 比服务端的小，就直接返回 -EOLDSNAPC 错误码。

3）如果是 read 操作，该对象的 ObjectContext 加 ondisk_read_lock 锁；对于源对象，无论读写操作，都需要加 ondisk_read_lock 锁。

所谓的源对象，就是一个操作中带两个对象，比如 copy 操作，源对象会有读操作。

4）调用函数 prepare_transaction 把相关的操作封装到 ctx->op_t 的事务中。如果是读操作，对于 replicate 类型，该函数直接调用 pgbackend->objects_read_sync 同步读取数据。如果是 EC，就把请求加入 pending_async_reads 完成异步读取操作。

5）解除操作 3 中加的相关的锁。

6）如果是读操作，并且 ctx->pending_async_reads 为空，说明是同步读取，调用 complete_read_ctx 完成读取操作，给客户端返回应答消息。如果是异步读取，就调用函数 ctx->start_async_reads 完成异步读取。读操作到这里就结束。后续都是写操作的流程。

7）调用 calc_trim_to，计算需要 trim 的 pg log 的版本。

8）调用函数 issue_repop 向各个副本发送同步操作请求。

9）调用函数 eval_repop，检查发向各个副本的同步操作是否已经 reply 成功，做相应的操作。

从上可以看出，execute_ctx 操作把相关的操作打包成事务，并没有真正的对对象的数据做修改。

8. calc_trim_to

本函数用于计算是否应该将旧的 pg log 日志进行 trim 操作：

```
void ReplicatedPG::calc_trim_to()
```

处理过程如下：

1）首先计算 target 值：target 值为最少保留的日志条数，默认设置为配置项 cct->_conf->osd_min_pg_log_entries 的值。如果 pg 处于 degraded，或者正在修复的状态，target 值为 cct->_conf->osd_max_pg_log_entries（默认 10000 条）。

2）变量 min_last_complete_ondisk 为本 pg 在本 osd 上的完成最后一条日志记录的版本。如果它不为空，且不等于 pg_trim_to，当前 pg log 的 size 大于 target 值，就计算需要 trim 掉的日志的条数：

a）num_to_trim 为日志总数目减去 target，如果它小于日志一次 trim 的最小值 cct->_conf->osd_pg_log_trim_min，就返回。

b）否则，从日志头开始计算最新的 pg_trim_to 版本。

9. prepare_transaction

本函数用于把相关的更新操作打包为事务，包括比较复杂的部分为对象的 snapshot 的处理：

```
int ReplicatedPG::prepare_transaction(OpContext *ctx)
```

处理过程如下：

1）首先调用函数 ctx->snapc.is_valid() 来验证 SnapSet 的有效性。

2）调用函数 do_osd_ops 打包请求到 ctx->op_t 的 transaction 中。

3）如果事务为空，或者没有修改操作，就直接返回 result。

4）检查磁盘空间是否满。

5）如果该对象是 head 对象，就有相关快照对象 COW 机制的操作，需要调用函数 make_writeable 来完成，在关于快照的介绍中会详细介绍到。

6）调用函数 finish_ctx 来完成后续处理，该函数主要完成了快照相关的处理。如果 head 对象存在，就删除 snapdir 对象；如果不存在，就创建 snapdir 对象，用来保存快照相关的信息。后文会进一步介绍。

10. issue_repop

```
void ReplicatedPG::issue_repop(RepGather *repop, OpContext *ctx)
```

本函数的处理过程如下：

1）首先更新 actingbackfill 的 osd 对应的 peer_info 的相关信息：如果 pinfo.last_update 和 pinfo.last_complete 二者相等，说明该 peer 的状态处于 clean 状态，就同时更新二者，否则只更新 pinfo.last_update 值。

2）对该对象的 ObjectContext 的 ondisk_write_lock 加写锁，如果有 clone 对象，对该 clone 对象的 ObjectContext 的 ondisk_write_lock 加写锁。如果 snapset_obc 不为空，也就是可能创建或者删除 snapdir 对象，对该 ObjectContext 的 ondisk_write_lock 加锁。

3）如果 pool 是可以 rollback 的（也就是 ErasureCode 模式），检查 pg log 也应该支持 rollback 操作。

4）分别设置三个回调 Context，调用函数 pgbackend->submit_transaction 来完成事务向从 osd 的发送。

本函数调用 pgbackend 的 submit_transaction 函数向从 osd 开始发送操作日志。

6.4.3　阶段 3：PGBackend 的处理

PGBackend 为 PG 的更新操作增加了一层与 PG 类型相关的实现。对于 Replicate 类型的 PG 由类 ReplicatedBackend 实现。其核心处理过程是把封装好的事务分发到该 PG 对应的其他从 OSD 上；对于 ErasureCode 类型的 PG 由类 ECBackend 实现，其核心处理过程为主 chunk 向各个分片 chunk 分发数据的过程。下面着重介绍 Replicate 的处理方式。

ReplicatedBackend::submit_transaction 函数最终调用网络接口，把更新的请求发送给从 OSD，其处理过程如下：

1）首先构建 InProgressOp 请求记录。

2）调用函数 ReplicatedBackend::issue_op 把请求发送出去：对于该 PG 中的每一个从 OSD：

a）调用函数 generate_subop 生成 MSG_OSD_REPOP 类型的请求。

b）调用函数 get_parent() → send_message_osd_cluster 把消息发送出去。

3）最后调用 parent → queue_transactions 函数来完成自己，也就是该 PG 的主 OSD 上本地对象的数据修改。

6.4.4　从副本的处理

当 PG 的从副本 OSD 接收到 MSG_OSD_REPOP 类型的操作，也就是主副本发来的同步写的操作时，处理流程和上述流程都一样。在函数 sub_op_modify_impl 里，对本地存储应用相应的事务，完成本地对象的数据写入。

6.4.5　主副本接收到从副本的应答

当 PG 的主副本接收到从副本的应答消息 MSG_OSD_REPOPREPLY 时，处理流程和上述类似，不同之处在于，在函数 ReplicatedPG::do_request 里调用了函数 ReplicatedBackend::handle_message，在该函数里调用了 ReplicatedBackend:: sub_op_modify_reply 函数处理该请求。

sub_op_modify_reply 函数的处理过程如下：

1）首先在 in_progress_ops 中查找到该请求。

2）如果是 ondisk 的 ACK，也就是事务已经应答，就在 ip_op.waiting_for_commit 删除该 OSD，该事务已经应答，那么必定已经提交了，那么从 ip_op.waiting_for_applied 删除该 OSD。

3）如果只是事务提交到日志中的 ACK，就从 ip_op.waiting_for_applied 删除。

这里特别说明的是，从副本需要给主副本发送两次 ACK，一次是事务提交到日志中，并没有应答到实际的对象数据中，一次是完成应答操作返回的 ACK。

4）最后检查，如果 ip_op.waiting_for_applied 为空，也就是所有从 OSD 的请求都返回来了，并且 ip_op.on_applied 不为空，就调用该 Context 的 complete 函数。同样，检查 ip_op.waiting_for_commit 为空，就调用该 Context 的函数的 complete 函数。

下面看一下，in_progress_ops 注册的回调函数。其回调函数是在 ReplicatedPG::issue_repop 函数调用里注册的。

```
Context *on_all_commit = new C_OSD_RepopCommit(this, repop);
Context *on_all_applied = new C_OSD_RepopApplied(this, repop);
Context *onapplied_sync = new C_OSD_OndiskWriteUnlock(
  repop->obc,
  repop->ctx->clone_obc,
  unlock_snapset_obc ? repop->ctx->snapset_obc : ObjectContextRef());
```

回调函数都最终调用了函数 ReplicatedPG::eval_repop，其最终向 client 发送应答消息。这里强调的是，主副本必须等所有的处于 up 的 OSD 都返回成功的 ACK 应答消息，才向客户端返回请求成功的应答。

6.5 本章小结

本章介绍了 OSD 读写流程核心处理过程。通过本章的介绍，可以了解读写流程的主干的流程，并对一些核心概念和数据结构的处理做了介绍。当然读写流程是 Ceph 文件系统的核心流程，其实现细节比较复杂，还需要读者对照代码继续研究。目前在这方面的工作，许多都集中在提供 Ceph 的读写性能。其基本的方法更多的就是优化读写流程的关键路径，通过减少锁来提供并发，同时简化一些关键流程。

第 7 章 Chapter 7

本地对象存储

本地对象存储模块完成了数据如何原子地写入磁盘，这就涉及事务和日志的概念。对象如何在本地文件系统中组织的代码实现在 src/os 中。本章将介绍在单个 OSD 上数据如何写入磁盘中。

目前有 4 种本地对象存储实现：

- FileStore：这是目前比较稳定，生产环境上使用的主流对象存储引擎，也是本章重点介绍的对象存储引擎。
- BlueStore：这是目前社区在实现的一个新版本，社区丢弃了本地文件系统，自己写了一个简单的，专门支持 RADOS 用户态的文件系统。
- KStore：这是以本地 KV 存储系统实现的对象存储，它基于 RODOS 的框架用来实现一个分布式的 KV 存储系统。
- Memstore：它把数据和元数据都保存在内存中，用来测试和验证使用。

KStore 和 Memtore 两种存储引擎比较简单，这里就不介绍了。BlueStore 社区还正在开发之中，这里也暂时不介绍。本章将详细介绍目前在生产环境中使用的 FileStore 存储的实现。

7.1 基本概念介绍

RADOS 本地对象存储系统（也称为对象存储引擎）基于本地文件系统实现，目前默认的文件系统为 XFS。一个对象包含数据和元数据两种数据。对应本地文件系统里，一个对象就是一个固定大小（默认 4MB）的文件，其元数据保存在文件的扩展属性或者本地独立的 KV 存储系统中。

7.1.1 对象的元数据

对象的元数据就是用于对象描述信息的数据，它以简单的 key-value（键值对）形式存在，在 RADOS 存储系统中有两种实现：xattrs 和 omap：

- xattrs 保存在对象对应文件的扩展属性中，这要求支持对象存储的本地文件系统支持扩展属性。
- omap 就是 object map 的简称，是一些键值对，保存在本地文件系统之外的独立的 key-value 存储系统中，例如 leveldb、rockdb 等。

有些本地文件系统可能不支持扩展属性，有些虽然也支持扩展属性但对 key 或者 value 占用空间的大小有限制，或者扩展属性占的总的空间大小有限制。对于 leveldb 等本地键值存储系统基本没有这样的限制，但是它的写性能优越于读性能。所以一般情况下，xattrs 保存一些比较小而经常访问的信息。omap 保存一些大而不是经常访问的数据。

7.1.2 事务和日志的基本概念

假设磁盘正在执行一个操作，此时由于发生磁盘错误，或者系统宕机，或者断电等其他原因，导致只有部分数据写入成功。这种情况就会出现磁盘上的数据有一部分是旧数据，部分是新写入的数据，使得磁盘数据不一致。

当一个操作要么全部成功，要么全部失败，不允许只有部分操作成功，就称这个操作具有原子性。引入事务和日志，就是为了实现操作的原子性，解决数据的不一致性问题。

引入日志后，数据写入变为两步：1）先把要写入的数据全部封装成一个事务，其整体作为一条日志，先写入日志文件并持久化的磁盘，这个过程称为日志的提交（journal

submit)。2）然后再把数据写入对象文件中，这称为日志的应用（journal apply)。

当系统在日志提交的过程中出错，系统重启后，直接丢弃不完整的日志条目即可，该条日志对应的实际对象数据并没有修改，数据可以保持一致。当在日志应用的过程中出错，由于数据已经写入并回刷到日志盘中，系统重启后，重放（replay）日志，就可以保证新数据重新完整写入，保证了数据的一致性。

这个机制需要确保所有的更新操作都是幂等操作。所谓幂等操作，就是数据的更新可以多次写入，不会产生任何副作用。对象存储的操作一般都具有幂等性。

在事务的提交过程中，一条日志记录可以对应一个事务。为了提高日志提交的性能，一般都允许多条事务并发提交，一条日志记录可以对应多个事务，批量提交。所以事务的提交过程，一般和日志的提交过程是一个概念。

日志有三个处理阶段，对应过程分别为：

- 日志提交（journal submit)：日志写入日志磁盘。
- 日志的应用（journal apply)：日志对应的修改更新到对象的磁盘文件中。这个修改不一定写入磁盘，可能缓存在本地文件系统的页缓存（page cache）中。
- 日志的同步（Jouranl sync 或者 journal commit)：当确定日志对应的修改操作已经刷回到磁盘中，就可以把相应的日志记录删除，释放出所占的日志空间。

7.1.3　事务的封装

ObjectStore 的内部类 Transaction 用来实现相关的事务。它有两种封装形式，一种是 use_tbl，事务把操作的元数据和数据都封装在 bufferlist 类型的 tbl 中：

```
bool use_tbl;      // 标记是否使用 tbl 来 encode/decode
bufferlist tbl;
```

另一种是不使用 tbl，把元数据操作以 struct Op 的结构体，封装在 op_bl 中，把操作相关的数据封装在 dataz_bl 中：

```
bufferlist data_bl;
bufferlist op_bl;
bufferptr op_ptr;  // 操作临时指针
```

由于结构体 op 里保存的是 coll_id 和 object_id，所以需要保存 coll_t 到 coll_id 的映射关系，以及 ghobject_t 和 object_id 的映射关系。从这种存储方式就可以看出，这种方式和前一种的区别，是在数据封装上实现了一种压缩。当事务中多个操作有相同的对象时，只保存一次 ghobject_t 结构体，其他情况只保存 index 来索引。

```
map<coll_t, __le32> coll_index;  //coll_t -> coll_id 的映射关系
map<ghobject_t, __le32, ghobject_t::BitwiseComparator> object_index;
    //ghobject_t  ->  object_id 的映射关系
__le32 coll_id;    // 当前分配的 coll_id 的最大值
__le32 object_id; // 当前分配的 object_id 的最大值
```

数据结构 TransactionData 记录了一个事务中有关操作的统计信息：

```
struct TransactionData {
  __le64 ops;     // 本事务中操作数目
        // 以下记录了事务中的带的数据最大的操作的：
  __le32 largest_data_len;    // 最大的数据长度
  __le32 largest_data_off;     // 在对象中的偏移
  __le32 largest_data_off_in_tbl; // 在 tbl 中的偏移
  __le32 fadvise_flags;        // 一些标志
}
```

一个事务中，对应如下三类回调函数，分别在事务不同的处理阶段调用。当事务完成相应阶段工作后，就调用相应的回调函数来通知事件完成。注意每一类都可以注册多个回调函数：

```
list<Context *> on_commit;
list<Context *> on_applied;
list<Context *> on_applied_sync;
```

on_commit 是事务提交完成之后调用的回调函数；on_applied_sync 和 on_applied 都是事务应用完成之后的回调函数。前者是被同步调用执行，后者是在 Finisher 线程里异步调用执行。

7.2 ObjectStore 对象存储接口

下面通过对象存储的接口说明和代码示例，可以了解对象存储的基本功能及如何使用这些功能，从而对对象存储有一个概要了解。

7.2.1　对外接口说明

类 ObjectStore 是对象存储系统抽象操作接口。所有的对象存储引擎都有继承并实现它定义的接口。下面列出一些有代表性的函数接口。

- objectStore 初始化相关的函数：
  - mkfs　创建 objectstore 相关的系统信息。
  - mount　加载 objectsotre 相关的系统信息。
  - statfs　获取 objectstore 系统信息。
- 获取属性已经 collection 相关的信息：
  - getattr　获取对象的扩展属性 xattr。
  - omap_get　获取对象的 omap 信息。
- queue_transactions 是所有 ObjectStore 更新操作的接口。更新相关的操作（例如创建一个对象，修改属性，写数据等）都是以事务的方式提交给 ObjectStore，该函数被重载成各种不同的接口。其参数为：
  - Sequencer *osr　用于保证在同一个 Sequencer 的操作是顺序的。
  - list<Transaction*>& tls　要提交的事务，或者事务的列表。
  - Context *onreadablehe 和 Context *onreadable_sync 这个函数分别对应事务的 on_apply 和 on_apply_sync，当事务 apply 完成之后，修改后的数据可以被读取了。
  - Context *ondisk　这个回调函数就是事务进行 on_commit 后的回调函数。

7.2.2　ObjectStore 代码示例

下面通过一段 ObjectStore 的测试代码，来展示 ObjectStore 的基本功能和接口使用：

1）首先调用 create 方法创建一个 ObjectStore 实例。参数分别为配置选项 CephContext，对象存储的类型 :filestore。对象存储的目录名、日志文件名如下：

```
ObjectStore *store_ = ObjectStore::create(g_ceph_context,
                                          string("filestore"),
                                          string("store_test_temp_dir"),
                                          string("store_test_temp_journal"));
```

2）创建并加载 ObjectStore 本地对象存储：

```
store_->mkfs();
```

```
store_->mount();
```

3）创建了一个 collection，并写数据到对象中。对象的任何写操作，都先调用事务的相关写操作。事务把相关操作的元数据和数据封装，然后调用 store 的 apply_transaction 来真正实现数据的更改。

```
ObjectStore::Sequencer osr("test");
coll_t cid;
ghobject_t hoid(hobject_t(sobject_t("Object 1", CEPH_NOSNAP)));
bufferlist bl;
bl.append("1234512345");
int r;

ObjectStore::Transaction t;     //创建一个事务
t.create_collection(cid, 0);    //创建一个 collection
t.write(cid, hoid, 0, bl.length(), bl);    //写对象数据
r = store->apply_transaction(&osr, std::move(t));
                                //事务的应用，真正实现数据的写入操作
store->umount();
r = store->mount();
```

7.3 日志的实现

在本地对象中，日志是实现操作一致性的机制。在介绍 Filestore 之前，首先需要了解 Journal 的机制。本节首先介绍 Journal 的对外接口，通过对外提供的功能接口，可以了解日志的基本功能。然后详细介绍 FileJournal 的实现。

7.3.1 Jouanal 对外接口

通过 Ceph 中的 test_filejournal.cc 的一段测试程序，可以比较清楚地了解 Joural 的外部接口，通过这些外部接口可以了解 Journal 的功能，以及如何使用 Jouarnal：

```
FileJournal j(fsid, finisher, &sync_cond, path, directio, aio);
j.create();
j.make_writeable();

bufferlist bl;
bl.append("small");
j.submit_entry(1, bl, 0, new C_SafeCond(&wait_lock, &cond, &done));
wait();
j.close();
```

通过上述代码可以了解到，日志的基本使用方法如下所示：

1）创建一个 FileJournal 类型的日志对象，其构造函数 path 为日志文件的路径。

2）调用日志的 create 方法创建日志，并调用函数 make_writeable 设置为可写状态。

3）在 bufferlist 中添加一段数据作为测试的日志数据，调用 submit_entry 提交一条日志。

4）等待日志提交完成。

5）关闭日志。

7.3.2　FileJournal

类 FileJournal 以文件（包括块设备文件看做特殊文件）做为日志，实现了 Journal 的接口。下面先介绍 FileJournal 的数据结构和日志的格式，然后介绍日志的三个阶段：日志的提交（journal submit）、日志的应用（journal apply）、日志的同步（journal sync 或者 commit、trim）及其具体的实现过程。

1. FileJournal 数据结构

FileJournal 数据结构如下：

```
struct write_item {
  uint64_t seq;                  //日志的序号
  bufferlist bl;                 //日志的内容
  uint32_t orig_len;             //日志的原始长度
  TrackedOpRef tracked_op;       //操作的跟踪记录

  write_item(uint64_t s, bufferlist& b, int ol, TrackedOpRef opref) :
    seq(s), orig_len(ol), tracked_op(opref) {
    bl.claim(b, buffer::list::CLAIM_ALLOW_NONSHAREABLE);
    }
  write_item() : seq(0), orig_len(0) {}
};

list<write_item> writeq;         //保存 write_item 的队列
Mutex writeq_lock;               //writeq 相关的锁
Cond writeq_cond;                //writeq 相关的条件变量
```

结构体 write_item 封装了一条提交的日志，seq 为提交日志的序号，bl 保存了提交的日志数据，也就是提交的事务或者事务的集合序列化后的数据。Journal 并不关心日志数据的内容，它认为一条日志就是一段写入的数据，只负责把数据正确写入日志盘。orig_

len 记录日志的原始长度，由于日志字节对齐的需要，最终写入的数据长度可能会变化。tracked_op 用于跟踪记录一些统计信息。

所有提交的日志，都先保存到 writeq 这个内部的队列中，writeq_lock 和 writeq_cond 是 writeq 相应的锁和条件变量：

```
struct completion_item {
  uint64_t seq;      // 日志的序号
  Context *finish;   // 完成后的回调函数
  utime_t start;     // 提交时间
  TrackedOpRef tracked_op;
  completion_item(uint64_t o, Context *c, utime_t s,
          TrackedOpRef opref)
    : seq(o), finish(c), start(s), tracked_op(opref) {}
  completion_item() : seq(0), finish(0), start(0) {}
};

Mutex completions_lock;
list<completion_item> completions;
```

结构体 completion_item 记录准备提交的 item，保存在列表 completions 中。由锁 completions_lock 保护。

```
off64_t write_pos;              // 日志的写的开始位置
off64_t read_pos;               // 日志读取的位置
uint64_t writing_seq;           // 本次正在写入的最大 seq

uint64_t last_committed_seq;    // 最后同步完成的 seq
bool write_stop;                // 写线程是否停止的标记

enum {
    FULL_NOTFULL = 0,
FULL_FULL = 1,
} full_state;                   // 日志的磁盘空间的状态

// 日志头部
struct header_t {
  enum {
    FLAG_CRC = (1<<0),
  };

  uint64_t flags;              // 日志的一些标志
  uuid_d fsid;                 // 文件系统的 id
  __u32 block_size;            // 日志文件或者设备的块大小
  __u32 alignment;             // 对齐
```

```
  int64_t max_size;              // 日志的最大 size
  int64_t start;                 // offset of first entry 日志条目的起始地址
  uint64_t committed_up_to;      // 已经提交的日志的 seq，等同于 journaled_seq
  uint64_t start_seq;            // 等同于日志的起始，last_commited_seq + 1
} header;
```

结构体 header_t 是日志文件的头信息，保存日志相关的全局信息。

```
deque<pair<uint64_t, off64_t> > journalq;
    // seq -> end 用于记录日志在日志文件中的结束偏移，用于日志删除时可以知道偏移
uint64_t writing_seq;    // 当前正在写入的最大的 seq
bool must_write_header;  // 是否必须写入一个 header，当日志同步完成后，就设置本标准，必
须写入一个 header，以持久化保存应日志同步而变化了的 header 结构

Mutex write_lock;
// 特别需要指出，以上数据结构都受 write_lock 的保护

Mutex  finisher_lock;
Cond  finisher_cond;
uint64_t journaled_seq;  // 已经提交成功的日志的最大 seq
```

2. 日志的格式

日志的格式如下：

```
entry_header_t  +  journal data  +  entry_header_t
```

每条日志数据的头部和尾部都添加了 entry_header_t 结构。此外，日志在每次同步完成的时候设置 must_write_header 为 true 时会强制插入一个日志头 header_t 的结构，用于持久化存储 header 中变化了的字段。日志的结构如图 7-1 所示。

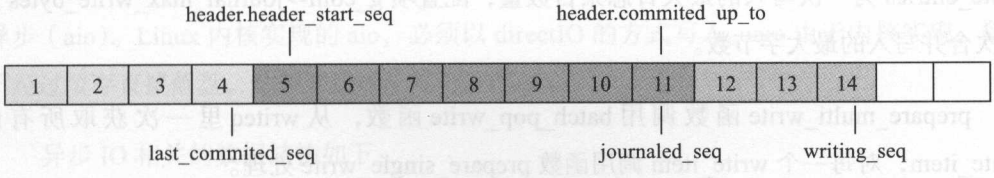


图 7-1　日志的结构图

日志记录 1 到 4 为已经由日志同步时 trim 掉的日志（逻辑删除），last_commited_seq 记录了最后一个 trim 的日志序号，当前值为 4。日志记录 5 到 11 为已经提交成功的日志。journaled_seq 记录已经提交成功的日志最大序号，当前值为 11。日志记录 12 到 14 为正在提交，还没有完成的日志。writing_seq 记录正在提交的最大日志序号，当前值为 14。

header_start_seq 记录提交成功的第一条有效日志，当前显示为 5，header.commited_up_to 为提交成功的最后一条日志，当前值为 11。

3. 日志的提交

函数 FileJournal::submit_entry 用于提交日志。

```
submit_entry
FileJournal::submit_entry(uint64_t seq, bufferlist& e, uint32_t orig_
len,Context *oncommit, TrackedOpRef osd_op)
```

其处理过程如下：

1）其分别加 writeq_lock 锁和 completions_lock 锁。

2）然后构造 completion_item 和 write_item 两个数据结构体，并分别加入到 completions 队列和 writeq 队列中。

3）如果 writeq 队列为空，就通过 writeq_cond.Signal() 触发写线程。（如果 writeq 不为空，说明线线程一直在忙碌处理请求，不会睡眠，就没必要再次触发了）。

write_thread 线程是 FileJournal 内部的一个工作线程。该线程的处理函数 write_thread_entry 实现了不断地从 writeq 队列中取出 write_item 请求，并完成把日志数据写入日志磁盘的工作。

write_thread_entry 函数调用 prepare_multi_write 函数，把多个 write_item 合并成到一个 bufferlist 中打包写入，可以提高日志写入磁盘的性能。配置项 g_conf->journal_max_write_entries 为一次写入的最大日志条目数量，配置项 g_conf->journal_max_write_bytes 为一次合并写入的最大字节数。

prepare_multi_write 函数调用 batch_pop_write 函数，从 writeq 里一次获取所有的 write_item，对每一个 write_item 调用函数 prepare_single_write 处理。

prepare_single_write 函数实现了把日志数据封装成日志格式的数据：

1）在数据的前端和后端都添加了一个 entry_header_t 的数据结构。

2）journalq 记录了该条日志在日志文件（或者设备）中的结束偏移量，它在日志同步时，用于逻辑删除该条日志：

```
journalq.push_back(pair<uint64_t,off64_t>(seq, queue_pos));
```

3）更新 writing_seq 的值为当前日志的 seq，并更新当前日志写入的 queue_pos 的值。当 qeueue_pos 达到日志的最大值时，需要从日志的头开始：

```
writing_seq = seq;
 queue_pos += size;
 if (queue_pos >= header.max_size)
 queue_pos = queue_pos + get_top() - header.max_size;
```

当调用函数 prepare_multi_write 把日志封装成需要的格式后，就需要把数据写入日志磁盘。目前有两种实现方式：一种是同步写入；另一种是 Linux 系统提供的异步 IO 的方式。

同步写入方式直接调用 do_write 函数实现，该函数具体过程如下：

1）首先判断是否需要写入日志的 header 头部：日志在每隔配置选项设定的 g_conf->journal_write_header_frequency 日志条数后或者 must_write_header 设置时插入一个 header 头。

2）其次计算如果日志数据超过了日志文件最大长度，就需要把该条日志的数据分两次写入：尾端写入一部分，然后绕到头部写入一部分。

3）如果不是 directIO，就需要调用函数 fsync 把日志数据刷到磁盘上。

4）加 finisher_lock，更新 journaled_seq 的值。

5）如果日志为不满的状态，并且 plug_journal_completions 没有设置，就调用函数 queue_completions_thru 来把 completion_item 的 callback 函数加入 Finisher。最后删除该 completion_item 项。

异步 IO 的方式写入是日志写入的另一种方式，是使用了 Linux 操作系统内核自带的异步（aio）。Linux 内核实现的 aio，必须以 directIO 的方式写入。aio 由于内核实现，数据不经过缓存直接落盘，性能要比同步的方式高很多。

异步 IO 相关的数据结构如下：

```
struct aio_info {          // 一条 aio 的记录信息
  struct iocb iocb;        // 提交的异步 aio 控制块
  bufferlist bl;           //aio 的数据
  struct iovec *iov;       // 异步 aio 方式的数据
  bool done;               // 是否完成的标志
  uint64_t off, len;       //< these are for debug only
  uint64_t seq;            //aio 的序号
};
```

```
io_context_t aio_ctx;          //异步 io 的上下文
list<aio_info> aio_queue;      //已经提交的 aio 请求队列

Mutex aio_lock;                //aio 队列的锁
Cond aio_cond;                 //控制 inflight 的 aio 不能过多，否则需要等待
Cond write_finish_cond;        //aio 完成，触发完成线程去处理

int aio_num, aio_bytes;        //正在进行的 aio 的数量和数据量
```

异步 IO 的方式写日志，调用了 do_aio_write 函数，基本过程和 do_write 相似，只是写入的方式调用了 write_aio_bl 函数。

函数 write_aio_bl 实现了调用 Linux 的 aio 的 API 异步写入磁盘。实现过程如下：

1）首先把数据转换成 iovec 的格式。

2）构造 aio_info 数据结构，该结构保存了 aio 的上下文信息，并把它加入 aio_queue 队列里。

3）调用函数 io_prep_pwritev 和函数 io_submit，异步提交请求：

```
io_prep_pwritev(&aio.iocb, fd, aio.iov, n, pos)
int r = io_submit(aio_ctx, 1, &piocb);
```

4）触发 write_finish_thread 检查是异步 IO 是否完成：

```
aio_lock.Lock();
write_finish_cond.Signal();
aio_lock.Unlock();
```

函数 write_finish_thread_entry 为线程 write_finish_thread 的执行函数，用来检查 aio 是否完成。其具体实现如下：

1）调用函数 io_getevents 来检查是否有 aio 事件完成的：

```
io_event event[16];
int r = io_getevents(aio_ctx, 1, 16, event, NULL);
```

2）对每个完成的事件，获取相应的 aio_info 结构。检查写入的数据是否是 aio 的数据长度，然后设置 ai → done 为 true，标记该 aio 已经完成。

3）调用函数 check_aio_completion 检查 aio_queue 队列的完成情况。由于 aio 的并发提交，多条日志可能并发完成，为了保证日志是按照 seq 的顺序完成，必须从头按顺序检查 aio_queue 的完成情况。

4）更新 journaled_seq 的值，调用 queue_completions_thru 完成后续工作，这和同步实现工作一样。

4. 日志的同步

日志的同步是在日志已经成功应用完成，对应的日志数据已经确保写入磁盘的日志后，就删除相应的日志，释放出日志空间的过程。

函数 committed_thru 用来实现日志的同步。它只是完成了日志的同步功能。日志同步时，必须确保日志应用完成，这个逻辑在 Filestore 里完成。

函数 committed_thru 具体实现如下：

1）确保当前同步的 seq 比上次 last_committed_seq 大。

2）用 seq 更新 last_committed_seq 的值。

3）把 jouralq 中小于等于 seq 的记录删除掉，这些记录已经无用了。

4）删除相应的日志。这里只是逻辑上的删除，只是修改 header.start 指针就可以了。当 journalq 为空，就删除到设置 write_pos 位置。如果 journalq 不为空，就设置为队列的第一条记录的偏移值，这是最新日志的起始地址，也就是上一条日志的结尾。

5）设置 must_write_header 为 true，由于 header 更新了，日志必须把 header 写入磁盘。

6）日志可能因日志空间满而阻塞写线程，此时由于释放出更多空间，就调用 commit_cond.Signal() 唤醒写线程。

7.4　FileStore 的实现

在介绍完本地对象存储的概念、对外接口和日志实现后，本节介绍 FileStore 的实现。首先看一下 FileStore 的静态类图，如图 7-2 所示。

FileStore 的静态类图说明如下：

- ObjectStore 是对象存储的接口。FileStore 继承了 JouralingObjectSore 类。
- JournalngObjectStore 实现了和日志之间的交互。

- FileStoreBackend 定义了本地文件系统支持 Filestore 的一些接口，不同的文件系统会增加自己支持特性实现了不同的 FileStoreBackend，例如 BtrfsFileStoreBackend 和 XfsFilestoreBackend 分别对应 XFS 和 BTRFS 两种文件系统的实现。
- Journal 类定义了日志的抽象接口，FileJoural 实现了以本地文件或者本地设备为存储的日志系统。

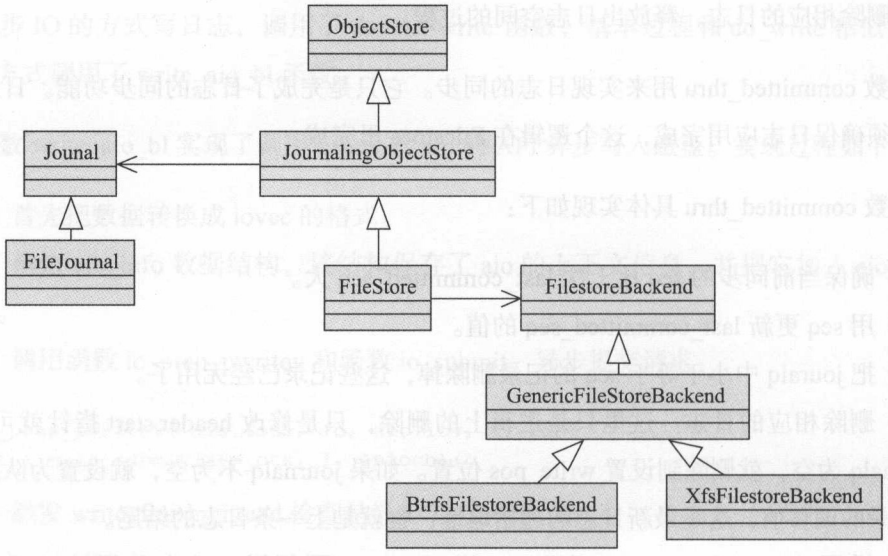


图 7-2 FileStore 的静态类图

7.4.1 日志的三种类型

在 FileStore 里，根据日志提交方式的不同，有三种类型的日志：

- journal writeahead：日志数据先提交并写入日志磁盘上，然后再完成日志的应用（更新实际对象数据）。这种方式适合 XFS、EXT4 等不支持快照的本地文件系统使用这种方式。
- journal parallel：日志提交到日志磁盘上和日志应用到实际对象中并行进行，没有先后顺序。这种方式适用于 BTRFS 和 ZFS 等实现了快照操作的文件系统。由于具有文件系统级别快照功能，当日志的应用过程出错，导致数据不一致的情况下，文件系统只需要回滚到上一次快照，并 replay 从上次快照开始的日志就可以了。显然，这种方式比 writeahead 方式性能更高。但是由于 btrfs 和 zfs 目前在 Linux 都不

稳定，这种方式很少用。

- 不使用日志的情况，数据直接写入磁盘后才返回客户端应答。这种方式目前 FileStore 也实现了，但是性能太差，一般不使用。

本节只介绍 FileStore 默认的方式：journal writeahead 方式的日志实现。

7.4.2 JournalingObjectStore

类 JournalingObjectStore 实现了 ObjectStore 和 FileJournal 的一些交互管理。它通过类 SubmitManager 和类 ApplyManager 分别实现了日志提交和日志应用的管理。下面分别介绍这两个类。

1. SubmitManager

类 SubmitManager 实现了日志提交的管理，准确地说是日志序号的管理。函数 op_submit_start 给提交的日志分配一个序号。函数 op_submit_finish 在日志提交完成后验证：当前的 op 的 seq 等于上次提交的序号 op_submit_finish 加 1。最后更新 op_submit_finish 的值。

```
class SubmitManager {
  Mutex lock;
  uint64_t op_seq;                  // 日志最新的序号
  uint64_t op_submit_finish;        // 日志上次提交的序号
 }
```

2. ApplyManager

ApplyManager 负责日志应用的相关处理：

```
class ApplyManager {
  Journal *&journal;
  Finisher &finisher;

  Mutex apply_lock;
  bool blocked;      // 日志应用是否阻塞，当前日志同步时需要
  Cond blocked_cond;
  int open_ops;      // 正在进行日志 apply 的 ops 的数量
  uint64_t max_applied_seq;    // 日志应用完成的最大的 seq

  Mutex com_lock;
  map<version_t, vector<Context*> > commit_waiters;
```

```
    // 日志完成后的回调函数，用于没有日志的类型
  uint64_t committing_seq,     // 正在应用的日志的 seq
           committed_seq;      // 已经应用完成的 seq
}
```

函数 ApplyManager::op_apply_start 在日志应用前调用，其实现如下操作：

1）给加 apply_lock 锁。

2）当 blocked 设置时，就等待，暂停日志的应用，这个后面日志同步时会说明。

3）统计值 open_ops 加 1。

函数 ApplyManager::op_apply_finish 在日志应用完成后调用，其实现如下操作：

1）加 apply_lock 锁。

2）统计值 open_ops 自减。

3）如果 blocked 为 true，就调用 blocked_cond.Signal() 发通知。

4）更新 max_applied_seq 的值为应用完成的最大日志号。

函数 ApplyManager::commit_start 用于在日志同步时计算目前完成的最大的日志 seq。

ApplyManager::commit_started 表示日志开始同步，commit_finish 函数在日志同步结束时调用。

7.4.3 Filestore 的更新操作

先介绍数据结构。结构 Op 代表了一个 ObjectStore 的操作的上下文信息在 FileStore 里的封装：

```
struct Op {
  utime_t start;    // 日志应用的开始时间
  uint64_t op;      // 日志的 seq
  list<Transaction*> tls;  // 事务列表
  Context *onreadable,*onreadable_sync;// 事务应用完成之后的回调函数和同步回调函数
  uint64_t ops, bytes;  // 操作数目和字节数，这里的 ops 指的是对象的基本操作例如 create,
                           write, delete 等基本操作，一个 Op 带多个 Transaction，可
                           能带多个基本操作。
    TrackedOpRef osd_op;
};
```

类 OpSequencer 用于实现请求的顺序执行。在同一个 Sequencer 类的请求，保证执

行的顺序，包括日志 commit 的顺序和 apply 的顺序。一般情况下，一个 PG 对应一个 Sequencer 类。所以一个 PG 里的操作都是顺序执行。

```
class OpSequencer{
  Mutex qlock; // 本来所保护队列 q，在 flush 时，peek 和 degueue 也被该锁保护
    list<Op*> q;          // 操作序列
    list<uint64_t> jq;    // 日志序号
    list<pair<uint64_t, Context*> > flush_commit_waiters;
    Cond cond;
  public:
    Sequencer *parent;
    Mutex apply_lock;     // 日志应用的互斥锁
    int id;
}
```

1. 更新实现

FileStore 的更新操作分两步：首先把操作封装成事务，以事务的形式整体提交日志并持久化到日志磁盘，然后再完成日志的应用，也就是修改实际对象的数据。

```
int FileStore::queue_transactions(Sequencer *posr, vector<Transaction>& tls,
      TrackedOpRef osd_op,
      ThreadPool::TPHandle *handle)
```

FileStore 更新的入口函数为 queue_transactions 函数。参数 posr 用于确保执行的顺序；参数 tls 是 Transaction 的列表，一次可以提交多个事务；TrackedOpRef 用于跟踪信息。

具体处理流程如下：

1）首先调用 collect_contexts 函数把 tls 中所有事务的 on_applied 类型的回调函数收集在一个 list<Context *> 结构里，然后调用 list_to_context 函数把 Context 列表变成了一个 Context 类。这里实际就是包装成一个回调函数。on_commit 和 on_applied_sync 都做了类似的工作。

2）构建 op 对象。把相关的操作封装到 op 对象里：

```
Op *o = build_op(tls, onreadable, onreadable_sync, osd_op);
```

3）调用函数 prepare_entry 把所有日志的数据封装到 tbl 里。

```
int orig_len = journal->prepare_entry(o->tls, &tbl);
```

4）分配一个日志的序号，并设置在 op 结构里。

```
uint64_t op_num = submit_manager.op_submit_start();
o->op = op_num;
```

5）如果日志是 m_filestore_journal_parallel 模式，就调用函数 _op_journal_transactions 开始提交日志。然后调用函数 queue_op，它把请求添加到 op_wq 里同时开始日志的应用。

6）如果日志是 m_filestore_journal_writeahead 模式，就把该 op 添加到 osr 中，调用 _op_journal_transactions 函数提交日志。

7）调用函数 submit_manager.op_submit_finish(op_num)，通知 submit_manager 日志提交完成。

2. writeahead 和 parallel 的处理方式的不同

函数 _op_journal_transactions 用于提交日志。当有日志并且日志可写时，就调用 journal 的 submit_entry 函数提交日志。当日志提交成功后，就会调用 _op_journal_transactions 注册的回调函数 onjournal。

函数 queue_op 用于把操作添加的 Filestore 内部的线程池对应队列中 OpWq 中。OpWq 的调用 _do_op 函数完成日志的应用，也就是完成实际的操作。

对于日志 parallel 方式，日志的提交和应用并发进行：

```
_op_journal_transactions(tbl, orig_len, o->op, ondisk, osd_op);
// 加入到提交队列里
queue_op(osr, o);
```

对于 writeahead 方式，先调用函数 _op_journal_transactions 提交日志，其注册的回调函数 onjournal 为 C_JourrnaledAhead：

```
osr->queue_journal(o->op);
_op_journal_transactions(tbl, orig_len, o->op,
  new C_JournaledAhead(this, osr, o, ondisk),
  osd_op);
```

在回调函数的 C_JournaledAhead 类里，其 finish 函数调用 _journaled_ahead，该函数调用 queue_op(osr, o) 把相应请求添加到 op_wq.queue(osr) 工作队列中。工作队列的处理线程实现了日志 apply 操作，完成实际对象数据的修改。

从上可以看出，writeahead 是先完成了日志的提交，然后才开始日志的应用。Parallel 方式是同时完成日志的提交和应用。

如果日志是第三种类型，即没有日志的形式，就调用函数 apply_manager.add_waiter 把 Contex 添加到 commit_waiters 中。

7.4.4　日志的应用

操作队列 OpWq 用于完成日志的应用。处理函数 _process 调用 _do_op 函数来完成应用日志，实现真正的修改操作。

函数 _do_op 函数实现操作如下：

1）首先调用 osr->apply_lock.Lock() 加锁，通过该锁，实现了同一时刻，osr 里只有一个操作在进行。

2）调用 op_apply_start，通知 apply_manager 类开始日志的应用。

3）调用 _do_transactions 函数，用于完成日志的应用。它对每一个事务调用 _do_transaction 函数，解析事务，执行相应的操作。

4）调用 op_apply_finish(o->op) 通知 apply_manager 完成。

7.4.5　日志的同步

在 FileStore 内部会创建 sync 线程，用来定期同步日志，该线程的入口函数为：

```
void FileStore::sync_entry()
```

函数 sync_entry 定期执行同步操，处理过程如下：

1）调用函数 tp.pause()，用来暂停 FileStore 的 op_wq 的线程池，等待正在应用的日志完成。

2）然后调用 fsync 同步内存中的数据到数据盘，当同步完成后，就可以丢弃相应的日志，释放相应的日志空间。

7.5 omap 的实现

omap 的静态类图 7-3 所示。类 ObjectMap 定义了 omap 的抽象接口。类 DBObjectMap 实现了以 KeyValueDB 的本地存储实现的 ObjectMap 的接口。

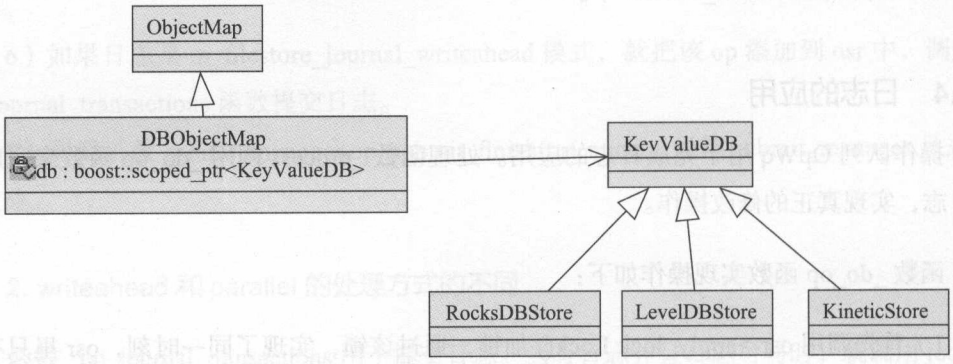


图 7-3 omap 静态类图

目前实现 KeyValueDB 的本地存储分别为：Facebook 开源的 levelDB 存储，对应类 LevelDBStore；Google 开源的 RocksDB 存储，对应类 RocksDBStore 实现；KineticStore 存储对应的类 kineticStore。默认采用 LevelDB 实现。

表 7-1 是 LevelDB 是一个 key-value 的存储系统，它是一维的 flat 模式的 KV 存储。表 7-2 是对象存储，多个不同对象的多个不同属性的二维存储模式。

二者如何映射呢？由于对象的名字是全局唯一的，属性在对象内也是唯一的，所以在 LevelDB 层面就可以用 Object 名字和属性的名字联合作为在 LevelDB 的 key，这是能想到的比较直观的解决方式。

例如：object1 的属性（key1，value1）的保存在 LevelDB 如下：

```
(object1_name + key1,   value1)
```

这种方法的一个缺点：当一个对象有多个 KV 值时，Object1 的 name 多次作为 key 存储，由于 Object 的 name 一般比较长，这样存储方式浪费空间比较大。于是就提出了一种压缩的存储方法，也就是目前 omap 的存储方式。

表 7-1　levelDB 的 flat 的 KV 存储模式

Key	Value
Key1	Value1
Key2	Value2
……	……
KeyN	Value N

表 7-2　对象的 omap 存储模式

Object1	Key1	Value1
	Key2	Value2
	……	……
Object2	Key1	Value1
	Key2	Value2
	……	……
……		

7.5.1　omap 存储

目前 omap 的在 leveldb 中存储分两步：

1）在 LevelDB 中，保存键值对：

```
key: HOBJECT_TO_SEQ + ghobject_key(oid)
value: header
```

HOBJECT_TO_SEQ 是固定的前缀标识字符串，函数 ghobject_key 获取对应的对象唯一的 key 字符串。

header 保存对象在 LevelDB 中的唯一标识 seq，以及支持快照的父对象的信息，同时保存了对象的 collection 和 oid（这里冗余保存，因为藏 key 信息里就可以获得）。

```
struct _Header {
  uint64_t seq;              // 自己在 leveldb 中的序号
  uint64_t parent;           // 父对象的序号
  uint64_t num_children;     // 子对象的数量
  coll_t c;                  // 对象所在的 collection
  ghobject_t oid;            // 对象标识
  SequencerPosition spos;    // 保存了当前的日志的序号（op_seq，日志内多个事务的序号，每个事务内操作的序号，
};
```

2）对象的属性保存以下格式的键值对：

```
Key:  USER_PREFIX + header_key(header->seq) + XATTR_PREFIX  + key
Value: value(omap 的值
```

综上所述，设置和获取对象的属性，需要两步：先根据对象的oid，构造键（HOBJECT_TO_SEQ + ghobject_key(oid)），获取 header；根据对象的 header 中的 seq，拼接在 levelDB 中的 key 值（USER_PREFIX + header_key(header->seq) + XATTR_PREFIX + key），获取 value 值。

变量 state 用于保存 KeyValueDB 的全局状态，目前只有 seq 信息。

```
struct State {
  __u8 v;          //版本
  uint64_t seq;    //全局分配的 seq
}state;
```

函数 write_state 用于每次分配 seq 后，把 state 信息写入 LevelDB 中，保存：

```
SYS_PREFIX + GLOBAL_STATE_KEY  ->  sate
```

7.5.2 omap 的克隆

omap 的克隆机制的实现如下所示：

```
oid        →  header
                 ↑  parent
oid_new    →  new_header
```

- 当克隆一个新的对象 old_new 时，仅创建一个对应的 new_header，并不是把该对象的所有属性都在 leveldb 中拷贝一遍，同时在 new_header 的 parent 字段保存 header 的 seq 号，从而建立了它们之间的父子联系。
- 当读取一个子对象的属性时，如果子对象不存在该属性，需要去父对象获取。

可以看出，omap 的 clone 机制也实现了 copy-on-write 机制。

7.5.3　部分代码实现分析

1. SequencerPosition

类 SequencerPosition 用来验证操作的顺序性，当操作执行时，后面的操作序号必须大于之前的操作序号。

```
struct SequencerPosition {
  uint64_t seq;   日志的序号
  uint32_t trans; // 一个日志内多个事务的序号
  uint32_t op;    // 一个事务内多个 op 的序号
}
```

2. lookup_create_map_header

本函数用于获取对象的 header：

1）首先调用函数 _lookup_map_header 查找对象的 header：

a）首先在 caches 里查找是否缓存。

b）调用底层 KeyValueDB 查找 header

```
int r = db->get(HOBJECT_TO_SEQ, map_header_key(oid), &out);
```

c）如果查找成功，就返回 header 对象，如果不成功，返回一个新创建的 header 对象。

2）调用函数 _generate_new_header 来设置 Header 的字段，并调用函数 write_state 写入全部变量 state：

```
header->seq = state.seq++;
if (parent) {
  header->parent = parent->seq;
  header->spos = parent->spos;
}
header->num_children = 1;
header->oid = oid;
```

3）调用函数 set_map_header，把新的 header 设置到 LevelDB 中。

3. get_xattrs

本函数用于获取对象的属性，实现如下：

1）首先获取对应的 Header 头部。

2）调用 db 设置具体的数据：

```
Header header = lookup_map_header(hl, oid);
if (!header)
  return -ENOENT;
return db->get(xattr_prefix(header), to_get, out);
```

4. set_keys

本函数用于设置对象的属性，实现如下：

1）获取 KeyValueDB::Transaction 的一个事务。

```
KeyValueDB::Transaction t = db->get_transaction();
```

2）先获取对象的 Header：

```
MapHeaderLock hl(this, oid);
Header header = lookup_create_map_header(hl, oid, t);
if (!header)
  return -EINVAL;
if (check_spos(oid, header, spos))
  return 0;
```

3）先调用事务 set 函数设置属性：

```
t->set(user_prefix(header), set);
```

4）调用 db 提交事务：

```
return db->submit_transaction(t)
```

7.6 CollectionIndex

Collection 的概念对应到本地文件系统中就是一个目录，用于存储一个 PG 里的所有的对象。

一个 collection 对应本地文件系统的一个目录，一个 PG 对应于一个 Collection，该 PG 的所有对象都保存在这个目录里，定义在类 coll_t 中：

```
class coll_t {
  enum type_t {
    TYPE_META = 0,
    TYPE_LEGACY_TEMP = 1,  // 这个类型不再使用
    TYPE_PG = 2,
    TYPE_PG_TEMP = 3,
  };

  type_t type ;    类型 meta, pg, temp
  spg_t pgid;      对应的 pgid
  uint64_t removal_seq;  // 这个字段不再使用，没有编码持久化存储
  string _str;  // 缓存的字符串
};
```

collection 有三种不同的类型：TYPE_META 类型表示这个 PG 里保存的是元数据（meta）相关的对象，TYPE_PG 表示该 collection 保存的是 PG 相关的数据，TYPE_PG_TEMP 保存临时对象。

当一个 PG 的对象数量比较多时，就会在一个目录里保存大量的文件。对于底层文件系统来说，如果一个目录里保存大量文件，当达到一定的程度后，性能会急剧下降。那么就需要一个 collection 里对应多个层级的子目录来存储大量文件，从而提高性能。

图 7-4 为 CollectionIndex 的静态类图。IndexManager 类为管理 CollecionIndex 的实现。HashIndex 实现了 LFNIndex，LFNIndex 实现了 CollectionIndex 接口。

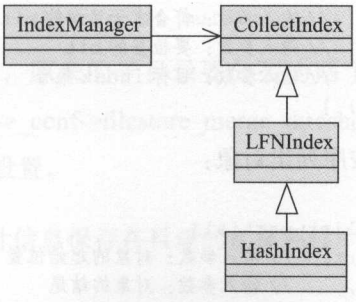


图 7-4　CollectionIndex 静态类图

7.6.1 CollectIndex 接口

CollectionIndex 使一个 Collection 里的对象保存在多层子目录中。类 CollectionIndex 是对象在文件系统中多层目录存储的接口。

通过接口说明，就可以看到其能提供的功能：

- 查找一个对象，返回对象对应文件的存储路径：

```
virtual int lookup(const ghobject_t &oid, // 输入参数，要查找的对象
  IndexedPath *path,                      // 输出参数，对象的路径
  int *hardlink
) = 0;
```

- 根据对象的路径，创建一个对象：

```
virtual int created(
  const ghobject_t &oid, // 创建对象名
  const char *path       // 创建对象路径
) = 0;
```

- 删除一个对象：

```
virtual int unlink(
  const ghobject_t &oid // 要删除的对象
) = 0;
```

- 分裂目录，当上次目录里保存的文件或者子目录达到一定数量，就需要分裂成两个目录：

```
virtual int split(
  uint32_t match,          // 输入参数：符合本子目录的 bit 值
  uint32_t bits,           // 输入参数：要检查的 bit
  CollectionIndex* dest    // 输入参数：目标 index 集合
) { assert(0); return 0; }
```

- 根据对象的 hash 值，按序列出对象：

```
virtual int collection_list_partial(
  const ghobject_t &start,  // 输入参数：对象的起始位置
  const ghobject_t &end,    // 输入参数：对象的结尾
  bool sort_bitwise,        // 输入参数：对象排序的顺序
  int max_count,            // 输入参数：最大对象数目
  vector<ghobject_t> *ls,   // 输出参数：对象列表
  ghobject_t *next          // 输出参数：下一次的对象起始
) = 0;
```

从上述接口介绍可知，CollectionIndex 提供了对象到其对应文件保存的目录路径映射管理。

7.6.2　HashIndex

HashIndex 是 CollectionIndex 的一个实现。HashIndex 实现了用对象的 Hash 值做为对象存储的目录。

1. 对象保存目录方式

以对象的 HASH 值为基准，从低位到高位十六进制的字符保存。

例如对象 ghobject_t("object", CEPH_NO_SNAP, 0xA4CEE0D2) 的 Hash 值是：0xA4CEE0D2，如果当前保存的目录层级是 2 级，那么该对象保存的目录为：

```
root/DIR_2/DIR_D
```

如果是 3 级目录，那么该对象保存的目录就是：

```
root/DIR_2/DIR_D/DIR_0
```

2. 目录层级

如何确定当前保存目录的层级呢？何时创建一个新的子目录呢？当一个目录中的对象数目超过如下值：

```
abs(merge_threshhold)) * 16 * split_multiplier
```

就重新创建一个子目录，原来的子目录要分裂为两个目录。其中 merge_threshhold 由配置选项 g_ceph_context->_conf->filestore_merge_threshold 设置。split_multiplier 由 g_conf->filestore_split_multiple 设置。

一个目录保存对象的统计信息保存在目录的扩展属性中，数据结构 subdir_info_s 定义了相关的属性；

```
struct subdir_info_s {
  uint64_t objs;        // 该目录中对象的数目
  uint32_t subdirs;     // 该目录中子目录的数目
```

```
    uint32_t hash_level; //子目录的 hash 层级数
}
```

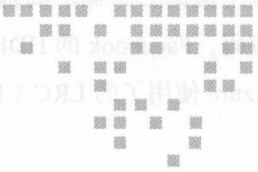
7.6.3 LFNIndex

HashIndex 继承了 LFNIndex 接口。LFNIndex 是 Long File Name Index 的缩写。从名称就可以知道，LFNIndex 用来处理如下情况：当对象名太长，超过了本地文件系统支持的长度时，LFXIndex 实现把超出的部分文件名保存到文件的扩展属性中。有可能保存到扩展属性的多个 key-value 存储对中。

7.7 本章小结

本章介绍本地对象存储的基本概念，以及 ObjectStore 对象存储接口，通过它可以了解对象存储如何调用。然后介绍了 Journal 的对外接口和 Filejournal 的实现，以及 Filestore 的更新操作。最后介绍了对象 omap 实现以及对象在本地文件系统中的组织方式。

目前对象存储是研究的热点。通过上述介绍可知，FileStore 的日志方式的写操作都需要数据的两次写入：一次写日志；另一次写数据对象。对于 S3 接口的对象存储等应用场景，双写是没有必要的。社区推出了新的存储引擎 BlueStore，其解决了某些场景下的避免双写，同时提高了性能。但目前 BlueStore 还不稳定，不能用于生产环境。



第 8 章 Chapter 8

Ceph 纠删码

本章介绍 Ceph 纠删码（Erasure Code，EC）的实现。首先介绍纠删码的原理，并对几种不同的编码原理进行分析，然后介绍纠删码的具体实现。其实现过程大部分逻辑和副本类似，这里着重介绍能完成数据回滚操作的不同实现点。

8.1 EC 的基本原理

纠删码（EC）是最近一段时间存储领域（特别是在云存储领域）比较流行的数据冗余存储的方法，它的原理和传统的 RAID 类似，但是比 RAID 方式更灵活。

它将写入的数据分成 N 份原始数据，通过这 N 份原始数据计算出 M 份效验数据。把 N+M 份数据分别保存在不同的设备或者节点中，并通过 N+M 份中的任意 N 份数据块还原出所有数据块。

EC 包含了编码和解码两个过程：将原始的 N 份数据计算出 M 份效验数据称为编码过程；通过这 N+M 份数据中的任意 N 份数据来还原出原始数据的过程称为解码过程。EC 可以容忍 M 份数据失效，任意小于等于 M 份的数据失效能通过剩下的数据还原出原始数据。

目前一些主流的云存储厂商都采用 EC 编码方式。Google GFS II 中采用了最基本的 RS（6，3）编码，Facebook 的 HDFS RAID 的早期编码方式为 RS（10，4）编码。微软的云存储系统 Azure 使用了的 LRC（12，2，2）编码。

8.2 EC 的不同插件

Ceph 支持以插件的形式来指定不同的 EC 编码方式。各种编码的不同点，实质就是在 ErasureCode 的三个指标之间折中的结果，这个三指标是：空间利用率、数据可靠性和恢复效率。

8.2.1 RS 编码

目前应用最广泛的纠删码是 ReedSolomon 编码，简称 RS 码。这种编码在 1960 年出现了，过了很多年以后才提出了快速算法并广泛应用于各种通信系统中，直到近几年才逐渐应用于各种存储系统中。下面介绍 RS 编码的几个实现。

1. Jerasure

Jerasure 是一个 ErausreCode 开源实现库，它实现了 EC 的 RS 编码。目前 Ceph 中默认的编码就是 Jerasure 方式。

2. ISA

ISA 是 Intel 提供的一个 EC 库，只能运行在 Intel CPU 上，它利用了 Intel 处理器本地指令来加速 EC 的计算。

RS 编码的不足之处在于：在 N+K 个数据块中有任意一块数据失效，都需要读取 N 块数据来恢复丢失数据。在数据恢复的过程中引起的网络开销比较大。因此，LRC 编码和 SHEC 编码分别从不同的角度做了相关优化。

8.2.2 LRC 编码

LRC 编码的核心思想为：将校验块（parity block）分为全局校验块（global parity）和

局部校验块（local reconstruction parity），从而减少恢复数据的网络开销。其目标在于解决当单个磁盘失效后恢复过程的网络开销。

LRC（M,G, L）的三个参数分别为：

- M 是原始数据块的数量。
- G 为全局校验块的数量。
- L 为局部校验块的数量。

编码过程为：把数据分成 M 个同等大小的数据块，通过该 M 个数据块计算出 G 份全局效验数据块。然后把 M 个数据块平均分成 L 组，每组计算出一个本地数据效验块，这样共有 L 个局部数据校验块。

下面以 Azure 的 LRC（12,2,2）和 Facebook 的 HDFS RAID 的早期编码方式 RS（10, 4）为例来比较 LRC 和 RS 编码在恢复过程的开销。参见表 8-1.

表 8-1　LRC 编码举例及其与 RS 的比较

LRC（12,2,2）	RS（12+4）
D_1 D_2 D_3 D_4 D_5 D_6 D_7 D_8 D_9 D_{10} D_{11} D_{12} L_1　　L_2 G_1　G_2	D_1 ~ D_{12} P_1 ~ P_4

表 8-1 所示对应 LRC 编码：总共有 12 个数据块，分别为 D_1 ~ D_{12}。有两个本地数据校验块 L_1 和 L_2，L_1 为通过第一组数据块 D_1 ~ D_6 计算而得的本地效验数据块；L_2 为第二组数据块 D_7 ~ D_{12} 计算而得的本地效验数据块。有 2 个全局数据效验块 G_1 和 G_2，它是通过所有数据块 D_1 ~ D_{12} 计算而来。对应 RS 编码，数据块为 D_1 ~ D_{12}，计算出的效验块为 P_1 ~ P_4。

不同情况下的数据恢复开销：

- 如果数据块 D_1 ~ D_{12} 只有一个数据块损坏，LRC 只需要读取 6 个额外的数据块来恢复。而 RS 需要读取 12 个其他的数据块来修复。
- 如果 L_1 或者 L_2 其中一个数据块损坏，LRC 需要读取 6 个数据块。如果 G_1，G_2 其中一个数据损坏，LRC 仍需要读取 12 个数据块来修复。

最大允许失效的数据块：

- RS 允许数据块和校验块中任意的小于等于 4 个数据的失效。
- 而对于 LRC：
  - 数据块中，只允许任意的小于等于 2 个数据块失效。
  - 允许所有的效验块（G_1，G_2，L_1，L_2）同时失效。
  - 允许至多两个数据块和两个本地效验块同时失效。

综上分析可知：对于只有一个数据块失效，或者一个本地数据效验块失效的情况下，在恢复该数据块时，LRC 比 RS 可以减少一半的磁盘 IO 和网络带宽。所以 LRC 重点在单个磁盘失效后恢复的优化。但是对于数据可靠性来说，通过最大允许失效的数据块个数的讨论可知，LRC 会有一定的损失。

8.2.3 SHEC 编码

SHEC 编码方式为 SHEC（K，M，L），其中 K 代表 data chunk 的数量，M 代表 parity chunk 的数量，L 代表计算 parity chunk 时需要的 data chunk 的数量。其最大允许失效的数据块为：ML/K。这样恢复失效的单个数据块只需要额外读取 L 个数据块。

下面以 SHEC（10，6，5）为例，其最大允许失效的数据块为：

```
M(6) * L(5)/ K(10 ) = 3
```

如图 8-1 所示的，D_1 ~ D_{10} 位数据块，P_1 为数据块 D_1 ~ D_5 计算出的校验块。P_2 位 D_3 ~ D_7 计算出的校验块。其他校验块的计算如图所示。当一个数据块失效时，只读取 5 个数据块就可以恢复。

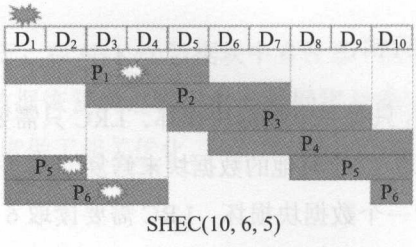


图 8-1 SHEC 编码示意图

8.2.4　EC 和副本的比较

在 Ceph 中可以设置一个 pool 为 EC 类型，并可以设置 N 和 M 的参数。副本和各种 RC 的编码比较如表 8-2 所示。

表 8-2　副本和各种 RC 的编码比较

	三副本	RS（10,4）	LRC（10,6,5）	SHEC（10,6,5）
数据容量开销	3X	1.4X	1.8X	1.6X
数据恢复开销（单个数据块失效）	1X	10X	5X	5X
可靠性	高	中	中	中下

说明如下：

- 在三副本的情况下，恢复效率和可靠性都比较高，缺点就是数据容量开销比较大。
- 对于 EC 的 RS 编码，和三副本比较，数据开销显著降低，以恢复效率和可靠性为代价。
- LRC 编码以数据容量开销略高的代价，换取了数据恢复开销的显著降低。
- SHEC 编码用可靠性换代价，在 LRC 的基础上进一步降低了容量开销。

8.3　Ceph 中 EC 的实现

8.3.1　Ceph 中 EC 的基本概念

下面介绍一些 EC 的基本概念。注意，这里 stripe 的概念是指 RADOS 系统定义的，可能与其他系统的定义不同。

- chunk：一个数据块就叫 data chunk，简称 chunk，其大小为 chunk_size 设置的字节数。
- stripe：用来计算同一个校验块的一组数据块，称为 data stripe，简称 stripe，其大小为 stripe_width，参与的数据块的数目为 stripe_size，这几个概念的关系如下：

  stripe_width = chunk_size × stripe_size

如图 8-2 所示为一个 EC（4+2）示例：stripe_size 为 4，chunk_size 的大小为 1K，那

么 stripe_width 的大小就为 4K。在 Ceph 系统中，默认的 stirpe_width 就为 4K。

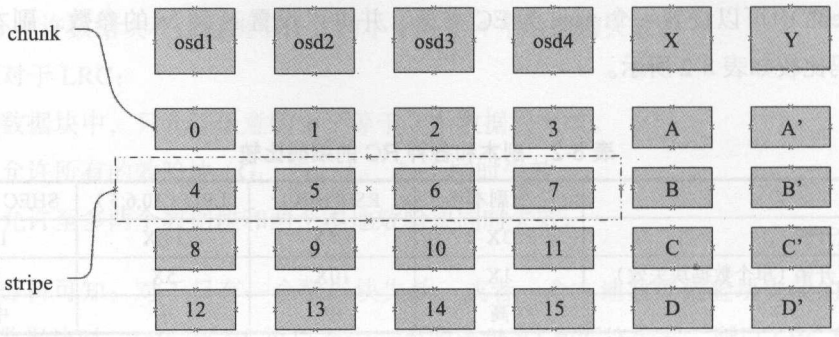


图 8-2 EC 的分片示意图

8.3.2 EC 支持的写操作

目前 Ceph 的 EC 方式的写入操作是有一定限制的，其目前只支持如下操作：

- create object：创建对象。
- remove object：删除对象。
- write full：写整个对象。
- append write（stripe width aligned）：追加写入（限定追加操作的起始偏移以 stripe_width 对齐）。

目前 Ceph 只支持上述操作，而不支持 overwrite 操作，其主要有如下两个条件的限制：

- 由于编码和解码的过程都以 stripe width 整块数据计算。
- EC 在特殊场景需要回滚的机制。

所以，目前 EC 只支持 append 写操作中，写操作的起始偏移 offset 以 stripe_width 对齐的情况，如果 end 不是以 stripe_width 对齐，就补 0 对齐即可。

目前不支持以下情况：

- **情况 1**：append 写操作，写操作的起始偏移 offset 没有以 stripe_width 对齐。
- **情况 2**：overwrite 写操作，offset 和 end 都不以 stripe_width 对齐。

由于计算数据校验块需要读取整个 stripe 的数据块。所在情况 1 和情况 2 都需要读取该 stripe 缺失的数据块，来计算效验块。由于性能的原因，目前不支持。

- **情况 3**：overwrite 写操作，写操作的起始偏移 offset 和结束位置 end 都以 stripe_width 对齐。

情况 3 目前也不支持，其原因是由 EC 的回滚的机制导致。下面将介绍 EC 的回滚机制。

8.3.3　EC 的回滚机制

依据 EC 的原理可知，EC（N+M）的写操作如果有小于等于 M 个 OSD 失效，不会导致数据丢失，数据可恢复。EC 在理论上就最多只能容忍 M 个 OSD 失效。如果 OSD 失效的数量大于 M，这种情况超出了理论设计的范畴，系统无法处理这种情况。可以说这是合理的。

但是对于所有的存储系统，必须应对一种特殊的情况：整个机房或者整个数据中心全局断电，系统重启后可恢复，并且数据不丢失。

当存储系统全局断电时，其数据的写入状态就有可能出现：小于 N 个磁盘的数据成功写入，而其他磁盘没有写成功的情况。

以图 8-2 所示的 EC（4+2）为例，假设写操作只有 3 个 OSD 写成功了，其他 3 个 OSD 没有来得及把数据写入磁盘。这种情况下，不但导致新数据写入失败，而且导致旧数据也无法读取成功。这就需要 EC 支持回滚机制，回滚到最后一次成功写入的旧数据版本。

Ceph 目前支持的 EC 操作都是回滚比较容易实现的，实现机制如下：

- create object 操作的回滚实现比较简单，删除该对象即可。
- 对于 remove object 操作，在执行时并不删除该对象，而是暂时保留该对象；如果需要回滚，就可以直接恢复。
- writeFull 操作：暂时保留旧的对象，创建一个新的对象完成写操作。当需要回滚时，恢复旧的数据对象。

- append 操作：记录 append 时的 size 到 PG 日志中；当回滚时，对该对象做 truncate 操作即可。

8.4 EC 的源代码分析

对应 EC 的上述三种更新操作，其本地回滚的信息都记录在对应的 PG 日志记录的 mod_desc 里：

```
struct pg_log_entry_t{
  ......
  ObjectModDesc mod_desc;
  ......
};
```

在函数 ReplicatedPG::do_osd_ops 中实现操作的事务封装，下面着重分析一下 EC 的写操作和 write_full 操作的实现。

8.4.1 EC 的写操作

操作步骤如下：

1）首先验证如果是 EC 类型，写操作的 offset 必须以 stripe_width 对齐，否则不支持。

```
case CEPH_OSD_OP_WRITE:
  if (pool.info.requires_aligned_append() &&
    (op.extent.offset % pool.info.required_alignment() != 0)) {
      result = -EOPNOTSUPP;
      break;
}
```

2）如果对象不存在，就在 mod_desc 中添加创建的信息，否则在 mod_desc 中添加 old size 的信息：

```
ctx->mod_desc.create();
```

否则就是追加写：

```
ctx->mod_desc.append(oi.size);
```

3）最后把写操作添加到事务中：

```
if (pool.info.require_rollback())
  t->append(soid, op.extent.offset, op.extent.length, osd_op.indata,
op.flags);
```

8.4.2　EC 的 write_full

操作步骤如下：

1）如果对象已经存在，调用函数 ctx->mod_desc.rmobject，如果返回 flase，说明已经记录了信息，直接删除；如果返回 true，就调用 stash 保存旧的对象数据，用来恢复：

```
case CEPH_OSD_OP_WRITEFULL:
  ......
  if (obs.exists) {
    if (ctx->mod_desc.rmobject(ctx->at_version.version)) {
      t->stash(soid, ctx->at_version.version);
    } else {
      t->remove(soid);
    }
  }
```

2）在事务中写入数据：

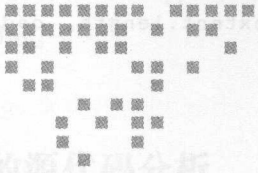
```
t->append(soid, 0, op.extent.length, osd_op.indata, op.flags);
```

8.4.3　ECBackend

类 ECBackend 实现了 EC 的读写操作。ECUtil 里定义了编码和解码的函数实现。ECTransaction 定了 EC 的事务。相关的代码都比较清晰，这里就不详细介绍了。

8.5　本章小结

本章介绍 Ceph 纠删码的编码原理，以及目前支持的操作和目前尚不支持其他操作的原因，并简单介绍了 EC 的代码实现。目前纠删码的研究是一个热点。它可以极大地提供存储利用率，降低存储成本。目前研究都在着力研究纠删码如何直接支持块存储，也就是随机 overwrite 操作的能力。



Chapter 9 第 9 章

Ceph 快照和克隆

本章介绍 Ceph 的高级数据功能：快照和克隆，它们在企业级的存储系统中是必不可少的。这里首先介绍 Ceph 中快照和克隆的基本概念，其次介绍快照实现相关的数据结构，然后介绍快照操作的原理，最后分析快照的读写操作的源代码实现。

9.1 基本概念

下面介绍快照和克隆的基本概念，以及二者之间的区别。

9.1.1 快照和克隆

快照是一个 RBD 在某一时刻全部数据的只读镜像。克隆是在某一时刻全部数据的可写镜像。快照和克隆都是某一时间点的镜像，区别在于快照只能读，而克隆可以写。

Ceph 支持两种类型的快照：一种 pool 级别的快照（pool snap），是给 pool 整体做一个快照；另一种是用户管理的快照（self managed snap）。目前 RBD 快照的实现就属于后者。用户的写操作必须自己提供 SnapContex 信息。注意，这两种快照是互斥的，两种快照不能同时存在。也就是说，如果对 pool 整体做了快照操作，就不能对该 pool 中的 RBD 做快

照操作。

无论是 pool 级别的快照，还是 RBD 的快照，其实现的基本原理都是相同的。都是基于对象的 COW（copy-on-write）机制。Ceph 可以完成秒级的快照操作和克隆操作。

这里需要特别指出的是，对象的 clone 操作指的是快照对应的克隆操作，是 RADOS 在 OSD 服务端实现的对象的拷贝。RBD 的 clone 操作是 RBD 的客户端实现的 RBD 层面的克隆。它们俩不是一个概念，希望读者区分开来。

在具体的实现过程中，克隆依赖快照的实现，克隆是在一个快照的基础上实现了可写功能。

图 9-1 是快照和克隆的示意图，其生成过程如下所示：

1）首先创建一个 RBD 块设备：rbd1。

2）对该块设备 rbd1 创建一个快照 snap1。

3）调用 rbd protect 来保护该快照，则该快照就不能被删除。

4）从该快照中克隆出一个新的 image，其名字为 clone1。

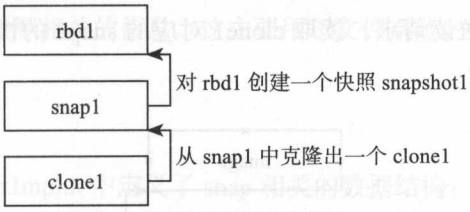


图 9-1　快照和克隆示意图

一个 image 的数据对象和快照对象都在同一个 pool 中，每个 image 的对象和对应的快照对象都在相同 OSD 上的相同 PG 中。快照的对象拷贝都是在 OSD 本地进行。

9.1.2　RBD 的快照和克隆比较

RBD 的快照和克隆在实现层面完全不同。快照是 RADOS 支持的，基于 OSD 服务端的 COW 机制实现的。而 RBD 的克隆操作完全是 RBD 客户端实现的一种 COW 机制，对于 OSD 的 Server 端是无感知的。

怎么理解 RBD 的克隆操作是由 RBD 的客户端实现的？如图 9-1 所示的快照和克隆，对克隆的 image 的读写过程如下：

1）当对克隆 image，也就是 clone1 发起写操作时，客户端对应的 OSD 发送正常的写请求。

2）OSD 返回给客户端应答，表明该 OSD 上对应的对象不存在。

3）客户端要发读请求到给克隆 image 的父 image，读取对应 snap 1 上的数据返回给客户端。

4）客户端把该快照数据写入克隆 image 中。

5）客户端给克隆 image 发送写操，写入实际要写入的数据。

由以上过程可知，克隆的拷贝操作是由客户端控制完成，OSD 的 Sever 端配合完成普通的读写操作。

当克隆的层级比较多时，需要客户端不断递归到其父 image 上去读取对应的快照对象，这会严重影响克隆的性能。

如图 9-2 所示，当读写 clone2 时，客户端首先给 clone2 发送写请求，如果对象不存在，就需要向 clone1 发送读请求，读取 clone1 对应的 snap2 的快照数据，并写入 clone2，然后完成实际的写入操作。

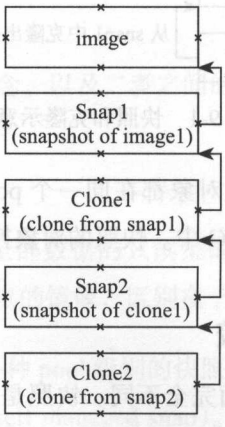


图 9-2　多层级的克隆

如果 clone1 上的对象不存在，同样，客户端继续递归，不断给给父 image 发送读请

求，读取 snap1 的快照数据，并写入 clone1 中。

如果层级过多就会影响克隆操作的性能。因此系统提供了 RBD 的 flattern 操作，可直接把父 image 的快照对象拷贝给克隆 image，这样以后就不需要去向父 image 查找对象数据了，从而提高了性能。

9.2　快照实现的核心数据结构

快照的核心数据结构如下：

- head 对象：也就是对象的原始对象，该对象可以进行写操作 。
- snap 对象：对某个对象做快照后，通过 cow 机制 copy 出来的快照对象只能读，不能写。
- snap_seq 或者 seq：快照序号，每次做 snapshot 操作系统都分配一个相应快照序号，该快照序号在后面的写操作中发挥重要作用。
- snapdir 对象：当 head 对象被删除后，仍然有 snap 和 clone 对象，系统自动创建一个 snapdir 对象，来保存 SnapSet 信息。head 对象和 snapdir 对象只有一个存在，其属性都可以保存快照相关的信息。这主要用于文件系统的快照实现。

1. SnapContext

在文件 librados/IoCtxImpl.h 中定义了 snap 相关的数据结构：

```
struct SnapContext {
  snapid_t seq;                  // 最新的快照序号
  vector<snapid_t> snaps;   // 当前存在的快照序号，降序排队
}
```

SnapContext 数据结构用来在客户端（RBD 端）保存 snap 相关的信息。这个结构持久化存储在 RBD 的元数据中：

```
struct librados::IoCtxImpl {
  ......
  snapid_t snap_seq;
  ::SnapContext snapc;
  ......
}
```

其中：

- seq 为最新的快照序号。
- snaps 降序保存了该 RBD 的所有的快照序号。

数据结构 IoCtxImpl 里的 snap_seq 一般也称为快照的 id（snap id）。当打开一个 image 时，如果打开的是一个卷的快照，那么 snap_seq 的值就是该 snap 对应的快照序号。否则 snap_seq 就为 CEPH_NOSNAP（-2），来表示操作的不是卷的快照，而是卷自身。

2. SnapSet

数据结构 SnapSet 用于保存 Server 端（也就是 OSD 端）与快照相关的信息：

```
struct SnapSet {
  snapid_t seq;                  // 最新的快照序号
  bool head_exists;              //head 对象是否存储
  vector<snapid_t> snaps;        // 所有的快照序号列表（降序排列）
  vector<snapid_t> clones;       // 所有的 clone 对象序号列表（升序排列）

  map<snapid_t, interval_set<uint64_t> > clone_overlap;
   // 和上次 clone 对象之间 overlap 的部分
  map<snapid_t, uint64_t> clone_size;
 //clone 对象的 size
}
```

下面是其中一些数据字段介绍：

- seq 保存最新的快照序号。
- head_exists 保存 head 对象是否存在。
- snaps 保存所有的快照序号。
- clones 保存所有快照后的写操作需要 clone 的对象记录。

这里特别强调的是 clones 和 snaps 的区别。由于不是每次做快照操作后，都要拷贝对象。只当快照操作后有写操作，才会触发相关对象的 clone 操作复制出一份新的对象，该对象是 clone 出来的，其快照序号记录在 clones 队列中，称为 clone 对象。

- clone_overlap 保存本次 clone 对象和上次 clone 对象（或者 head 对象）的 overlap 的部分，也就是重叠的部分。clone 操作后，每次写操作，都要维护这个信息。这个信息用于在数据恢复阶段对象恢复的优化。

- clone_size 保存每次 clone 后的对象的 size。

SnapSet 数据结构持久化保存在 head 对象的 xattr 的扩展属性中：

- 在 Head 对象的 xattr 中保存 key 为 snapset，value 为 SnapSet 结构序列化后的值。
- 在 snap 对象的 xattr 中保存 key 为 user.cephos.seq 的 snap_seq 值。

9.3　快照的工作原理

9.3.1　快照的创建

RBD 快照创建的基本步骤如下：

1）向 Monitor 发送请求，获取一个最新的快照序号 snap_seq 的值。

2）把该次快照的 snap_name 和 snap_seq 的值保存到 RBD 的元数据中。

在 RBD 的元数据里保存了所有快照的名字和对应的 snap_seq 号，并不会触发 OSD 端的数据操作，所以非常快。

9.3.2　快照的写操作

当对一个 image 做了一次快照后，该 image 写入数据时，由于快照的存在需要启动 copy-on-write（cow）机制。下面将介绍 cow 机制的具体实现。

客户端的每次写操作，消息中都必须带数据结构 SnapContex 信息，它包含了客户端认为的最新快照序号 seq，以及该对象的所有快照序号 snaps 的列表。在 OSD 端，对象的 Snap 相关信息保存在 SnapSet 数据结构中，当有写操作发生时，处理过程按照如下规则进行。

规则 1

如果写操作所带 SnapContex 的 seq 值小于 SnapSet 的 seq 值，也就是客户端最新的快照序号小于 OSD 端保存的最新的快照序号，那么直接返回 -EOLDSNAP 错误。

Ceph 客户端始终保持最新的快照序号。如果客户端不是最新的快照序号，可能的情

况是：在多个客户端的情形下，其他客户端有可能创建了快照，本客户端有可能没有获取到最新的快照序号。

Ceph 有一套 Watcher 回调通知机制来实现快照序号的更新。如果其他客户端对一个卷做了快照，就会产生了一个最新的快照序号。OSD 端接收到最新快照序号变化后，通知相应的连接客户端更新最新的快照序号。如果有客户端没有及时更新，也没有太大的问题，OSD 端会返回客户端 -EOLDSNAP，客户端会主动更新为最新的快照序号，重新发起写操作。

规则 2

如果 head 对象不存在，创建该对象并写入数据，用 SnapContext 相应的信息更新 SnapSet 的信息。

规则 3

如果写操作所带 SnapContex 的 seq 值等于 SnapSet 的 seq 值，做正常的读写。

规则 4

如果写操作所带 SnapContex 的 seq 值大于 SnapSet 的 seq 值：

1）对当前 head 对象做 copy 操作，clone 出一个新的快照对象，该快照对象的 snap 序号为最新的序号，并把 clone 操作记录在 clones 列表里，也就是把最新的快照序号加入到 clones 队列中。

2）用 SnapContex 的 seq 和 snaps 值更新 SnapSet 的 seq 和 snaps 值。

3）写入最新的数据到 head 对象中。

9.3.3 快照的读操作

快照读取数据时，输入参数为 RBD 的名字和快照的名字。RBD 的客户端通过访问 RBD 的元数据，来获取快照对应的 snap_id，也就是快照对应的 snap_seq 值。

在 OSD 端，获取 head 对象保存的 SnapSet 数据结构。然后根据 SnapSet 中的 snaps

和 clones 值来计算快照所对应的正确的快照对象。

9.3.4　快照的回滚

快照的回滚，就是把当前的 head 对象回滚到某个快照对象。具体操作如下：

1）删除当前 head 对象的数据。

2）拷贝相应的 snap 对象到 head 对象。

其源代码的实现在 ReplicatedPG::_rollback_to 里。

9.3.5　快照的删除

删除快照时，直接删除 rbd 的元数据中保存的 Snap 相关快照信息，然后给 Monitor 发快照删除信息。Monitor 随后给相应的 OSD 发送删除的快照序号，然后由 OSD 控制删除本地相应的快照对象。该快照是否被其他快照对象共享。

由上可知，Ceph 的快照删除是延迟删除，并不是直接立即删除。

9.4　快照读写操作源代码分析

下面着重分析快照的写操作和读操作相关的源代码实现。

9.4.1　快照的写操作

在结构体 OpContext 的上下文中，保存了快照相关的信息：

```
struct OpContext {
  const SnapSet *snapset;   // 旧的 SnapSet，也就是 OSD 服务端保存的快照信息。

  ObjectState new_obs;
  SnapSet new_snapset;      // 新的 SnapSet
  SnapContext snapc;        // 写操作带的，也就是客户端的 SnapContext 信息
};
```

在读写的关键流程中，有关快照的处理如下：

1）在 OSD 写操作的流程中，在函数 ReplicatedPG::execute_ctx 中，把消息带的 SnapContext 信息保存在了 OpContext 的 snapc 中：

```
ctx->snapc.seq = m->get_snap_seq();
ctx->snapc.snaps = m->get_snaps();
```

2）在 OpContext 的构造函数里，用结构 snapset 字段初始化了结构 new_snapset 的相关字段。当前 new_snapset 保存的就是 OSD 服务端的快照信息：

```
if (obc->ssc) {
   new_snapset = obc->ssc->snapset;
   snapset = &obc->ssc->snapset;
}
```

3）在函数 ReplicatedPG::prepare_transaction 里调用了函数 ReplicatedPG::make_writeable 来完成快照相关的操作。

9.4.2 make_writeable 函数

函数 make_writeable 处理快照相关的写操作，其处理流程如下：

1）首先判断，如果服务端的最新快照序号大于客户端的快照序号，就用服务端的快照信息更新客户端的快照信息：

```
if (ctx->new_snapset.seq > snapc.seq) {
   snapc.seq = ctx->new_snapset.seq;
   snapc.snaps = ctx->new_snapset.snaps;
   dout(10) << " using newer snapc " << snapc << dendl;
}
```

在数据读写的流程中可知，在函数 ReplicatedPG::execute_ctx 里已经判断了：客户端的最新快照序号不能小于服务端的快照序号，否则就直接返回 -EOLDSNAPC 错误码客户端更新快照序号后重试。所以笔者认为这段代码不会进入，所以是无用的代码。

2）调用函数 filter_snapc 把已经删除的快照过滤掉。

3）如果 head 对象存在，并且 snaps 的 size 不为空（有快照），并且客户端的最新快照序号大于服务端的最新快照序号，在这种情况下要克隆对象，实现对象数据的拷贝了：

a）构造 clone 对象 coid，其 coid.snap 为最新的客户端 seq 值。

b）计算 snaps 列表，也就是本次克隆对象对应的所有快照。

c）构造 clone_obc，也就是克隆对象 coid 的 ObjectContext。特别需要指出的是该克隆对象的 object_info_t 中的 snaps 信息，就是在上一步中计算出的 snaps 列表。

d）调用函数 _make_clone 实现对象的克隆操作。此时克隆操作都先封装在新创建的事务 t 中：

```
PGBackend::PGTransaction *t = pgbackend->get_transaction();
_make_clone(ctx, t, ctx->clone_obc, soid, coid, snap_oi);
t->append(ctx->op_t);
delete ctx->op_t;
ctx->op_t = t;
```

注意，之前的写操作封装在事务 ctx->op_t 中，把该事务追加到事务 t 的尾部，然后删除 ctx->op_t 事务，事务 t 赋值给 ctx->op_t。这样在事务应用时，就是先做克隆操作，然后才完成写操作。

4）最后把该克隆对象添加到 ctx->new_snapset.clones 中，并添加 clone_size 记录和 clone_overlap 记录。

5）根据当前的写操作修改范围 modified_ranges，来计算修改 clone_overlap 的记录，也就是当前 head 对象和上次克隆对象的重叠区域，该信息用来优化快照对象的恢复。

6）更新服务端的快照信息为客户端的快照记录信息。

下面举例说明。

例 9-1　快照写操作见表 9-1。

表 9-1　写操作示例

操作序列	操作	RBD 的元数据信息	OSD 端对应的对象

1	write data1
	snapContext{
	seq=0,
	snaps={}
	}
			SnapSet={
			seq = 0,
			head_exists= true,
			snaps={},
			clones={},
			}
			obj1_head(data1)

2	create snapshot name=" snap1"	("snap1",1)	

（续）

操作序列	操作	RBD 的元数据信息	OSD 端对应的对象

3	write data2
	snapContext{
	seq=1,
	snaps={1}
	}
			SnapSet={
			seq = 1,
			head_exists= true,
			snaps={1},
			clones={1},
			}
			obj1_1(data1)
			obj1_head(data2)

4	create snapshot name=" snap3"	("snap1",1) ("snap3",3)	
5	create snapshot name=" snap6"	("snap1",1) ("snap3",3) ("snap6",6)	

6	write data3
	snapContext{
	seq=6,
	snaps={6,3,1}
	}
			SnapSet={
			seq = 6,
			head_exists= true,
			snaps={6,3,1},
			clones={1,6},
			}
			obj1_1(data1)
			obj1_6(data2)
			obj1_head(data3)

说明如下：

1）在操作 1 里为第一次写操作，写入的数据为 data1，SnapContext 的初始 seq 为 0，snaps 列表为空。按规则 2，OSD 端创建对象并写入对象数据，用 SnapContext 的数据更新 SnapSet 中的数据。

2）在操作 2 里，创建了该 RBD 一个快照，名字为 snap1，并向 Monitor 申请分配一个快照序号，其值为 1。在该卷的元数据里添加了快照的名字和对应的快照序号。

3）操作 3 里，写入数据 data2，写操作所带 SnapContext 中的 seq 值为 1，snap 列表为 {1}。在 OSD 端处理，此时 SnapContext 的 seq 大于 SnapSet 的 seq，操作按照规则 4：

a）更新 SnapSet 中的 seq 为 1，snaps 列表更新为 {1} 值。

b）创建快照对象 obj1_1，拷贝当前 head 对象的数据 data1 到快照对象 obj1_1 中（快照对象名字下划线后面为快照序号，Ceph 目前快照对象的名字中含有快照序号）。此时快照对象 obj1_1 的数据为 data1，并在 clones 中添加 clone 操作记

录，clone 列表的值为 {1}。

c）向 head 对象 obj1_head 中写入数据 data2 中。

4）操作 4 和操作 5 连续做了两次快照操作，快照的名字分别为 snap3 和 snap6，分配的快照序号分别为 3,6（在 Ceph 里，快照序号是由 Monitor 分配的，全局唯一，所以单个 RBD 的快照序号不一定连续）。

5）操作 6 写入数据 data3，此时写操作所带 SnapContext 中的 seq 值为 6，snaps 值为 {6,3,1} 共三个快照。此时 SnapSet 的 seq 为 1，操作按规则 4 处理过程如下：

a）更新 SnapSet 结构中的 seq 值为 6，snaps 值为 {6,3,1}。

b）创建快照对象 obj1_6，拷贝当前 head 对象的数据 data2 到快照对象 obj1_6 中，并把本次克隆操作记录添加到 clone 队列中。更新后的 clone 队列的值为 {1,6}。

c）向 head 对象 obj1_head 中写入数据 data3。

9.4.3　快照的读操作

快照的读取操作核心在函数 ReplicatedPG::find_object_context 里实现，其原理是根据读对象的快照序号，查找实际对应的克隆对象的 ObjectContext。基本的步骤如下：

1）如果对象的快照序号 oid.snap 大于服务端的最新快照序号 ssc->snapset.seq，获取 head 对象就是该快照对应的实际数据对象。

2）计算 oid.snap 首次大于 ssc->snapset.clones 列表中的克隆对象，就是 oid 对应的克隆对象。

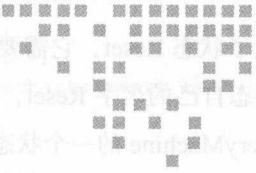
例如在例 9-1 中，最后的 SnapSet 为：

```
SnapSet = {
  seq=6,
  snaps={6,3,1},
  clones={1,6},
  ......
}
```

这时候读取 seq 为 3 的快照，由于 seq 为 3 的快照并没有写入数据，也就没有对应的克隆对象，通过计算可知，seq 为 3 的快照和 snap 为 1 的快照对象数据是一样的，所以就读取 obj1_1 对象数据。

9.5 本章小结

Ceph 的基于 Copy-on-Write 的机制实现了秒级别的快照，其高效率的核心原理在于做快照操作时不会直接拷贝数据，而是只做了快照的记录，当只有实际的写操作发生时，才实现对象的拷贝操作。实质就是把整个卷的拷贝操作开销分散到后续每次写操作过程中，这样就实现了快照操作只是增加了新的快照记录，所以快照操作可以在秒级实现。



第 10 章 Chapter 10

Ceph Peering 机制

本章介绍 Ceph 中比较复杂的模块：Peering 机制。该过程保障 PG 内各个副本之间数据的一致性，并实现 PG 的各种状态的维护和转换。本章首先介绍 boost 库的 statechart 状态机基本知识，Ceph 使用它来管理 PG 的状态转换。其次介绍 PG 的创建过程以及相应的状态机创建和初始化。然后详细介绍 Peering 机制三个具体的实现阶段：GetInfo、GetLog、GetMissing。

10.1 statechart 状态机

Ceph 在处理 PG 的状态转换时，使用了 boost 库提供的 statechart 状态机。因此先简单介绍一下 statechart 状态机的基本概念和涉及的相关知识，以便更好地理解 Peering 过程中 PG 的状态机转换流程。下面在举例时截取了 PG 状态机的部分代码。

10.1.1 状态

在 statechart 里，一个状态的定义方式有两种：

- 没有子状态情况下的状态定义：

```
struct Reset : boost::statechart::state< Reset, RecoveryMachine >
```

这里定义了状态 Reset，它需要继承 boost::statechart::state 类。该类的模板参数中，第一个参数为状态自己的名字 Reset，第二个参数为该状态所属状态机的名字，表明 Reset 是状态机 RecoveryMachine 的一个状态。

❑ 有子状态情况下的状态定义：

```
struct Started : boost::statechart::state< Started, RecoveryMachine, Start >
```

状态 Started 也是状态机 RecoveryMachine 的一个状态，模板参数中多一个参数 Start，它是状态 Started 的默认初始子状态，其定义如下：

```
struct Start : boost::statechart::state< Start, Started >
```

这里定义的 Start 是状态 Started 的子状态。第一个模板参数是自己的名字，第二个模板参数是该子状态所属父状态的名字。

综上所述，一个状态，要么属于一个状态机，要么属于一个状态，成为该状态的子状态。其定义的模板参数，第一个参数是自己，第二个参数是拥有者，第三个参数是它的起始子状态。

10.1.2 事件

状态能够接收并处理事件。事件可以改变状态，促使状态发生转移。在 boost 库的 statechart 状态机中定义事件的方式如下所示：

```
struct QueryState : boost::statechart::event< QueryState >
```

QueryState 为一个事件，需要继承 boost::statechart::event 类，模板参数为自己的名字。

10.1.3 状态响应事件

在一个状态内部，需要定义状态机处于当前状态时可以接受的事件以及如何处理这些事件的方法：

```
struct Initial : boost::statechart::state< Initial, RecoveryMachine >, NamedState {
 typedef boost::mpl::list <
```

```
    boost::statechart::transition< Initialize, Reset >,
    boost::statechart::custom_reaction< Load >,
    boost::statechart::custom_reaction< NullEvt >,
    boost::statechart::transition< boost::statechart::event_base, Crashed >
  > reactions;

    boost::statechart::result react(const Load&);
    boost::statechart::result react(const MNotifyRec&);
    boost::statechart::result react(const MInfoRec&);
    boost::statechart::result react(const MLogRec&);
              boost::statechart::result
              react(const boost::statechart::event_base&) {
      return discard_event();
    }
}
```

上述代码列出了状态 RecoveryMachine/Initial 可以处理的事件列表和处理对应事件方法：

- 通过 boost::mpl::list 定义该状态可以处理多个事件类型。在本例中可以处理 Initialize、Load、NullEvt 以及 event_base 事件。
- 简单事件处理：

```
boost::statechart::transition< Initialize, Reset >
```

定义了状态 Initial 接收到事件 Initialize 后，无条件直接跳转到 Reset 状态。

- 用户自定义事件处理：当接收到事件后，需要根据一些条件来决定状态如何转移，这个逻辑需要用户自己定义实现：

```
boost::statechart::custom_reaction< Load >
```

custom_reaction 定义了一个用户自定义的事件处理方法，必须有一个 react 的处理函数处理对应该事件。状态转移的逻辑需要用户自己在 react 函数里实现：

```
boost::statechart::result react(const Load&)
```

- NullEvt 事件用户自定义处理，但是没有实现 react 函数来处理，最终事件匹配了 boost::statechart::event_base 事件，直接调用函数 discard_event 把事件丢弃掉。

10.1.4　状态机的定义

RecoveryMachine 为定义的状态机，需要继承 boost::statechart::state_machine 类。

```
class RecoveryMachine : public boost::statechart::state_machine< Recovery-
  Machine, Initial >
```

模板参数第一个参数为自己的名字，第二个参数为状态机默认的初始状态 Initial。

状态机的基本操作有两个：

❑ 状态机的初始化：

```
machine.initiate()
```

❑ 函数 process_event 用来向状态机投递事件，从而触发状态机接收并处理该事件：

```
machine.process_event(evt);
```

10.1.5 context 函数

context 是状态机的一个比较有用的函数，它可以获取当前状态的所有祖先状态的指针。通过它可以获取父状态以及祖先状态的一些内部参数和状态值。

例如状态 Start 是 RecoveryMachine 的一个状态，状态 Started 是 Start 状态的一个子状态，那么如果当前状态是 Started，就可以通过该函数获取它的父状态 Start 的指针：

```
Start* parent  =  context<Start>()
```

同时也可以获取其祖先状态 RecoveryMachine 的指针：

```
RecoveryMachine*  grandparent = context<RecoveryMachine>()
```

综上所述，context 函数为获取当前状态的祖先状态上下文提供了一种方法。

10.1.6 事件的特殊处理

事件除了在状态转移列表中触发状态转移，或者进入用户自定义的状态处理函数，还可以有下列特殊的处理方式：

❑ 在用户自定义的函数里，可以直接调用函数 transit 来直接跳转到目标状态。例如：

```
transit<WaitRemoteBackfillReserved>()
```

可以直接跳转到状态 WaitRemoteBackfillReserved。

- 在用户自定义的函数里，可以调用函数 post_event 直接产生相应的事件，并投递给状态机。
- 在用户自定义的函数里，调用函数 discard_event 可以直接丢弃事件，不做任何处理。
- 在用户自定义的函数里，调用函数 forward_event 可以把当前事件继续投递给状态机。

10.2　PG 状态机

在类 PG 的内部定义了类 RecoveryState，该类 RecoveryState 的内部定义了 PG 的状态机 RecoveryMachine 和它的各种状态。

在每个 PG 对象创建时，在构造函数里创建一个新的 RecoveryState 类的对象，并创建相应的 RecoveryMachine 类的对象，也就是创建了一个新的状态机。每个 PG 类对应一个独立的状态机来控制该 PG 的状态转换。

图 10-1 为 PG 状态机的总体状态转换图，相对比较复杂，在介绍相关的内容模块时再逐一详细介绍。

10.3　PG 的创建过程

在 PG 的创建过程中完成了 PG 对应的状态机的创建和状态机的初始化操作。一个 PG 的创建过程会由其所在 OSD 在 PG 中担任的角色不同，创建的机制也不相同。

10.3.1　PG 在主 OSD 上的创建

当创建一个 Pool 时，通过客户端命令行给 Monitor 发送创建 Pool 的命令，Monitor 对该 Pool 的每一个 PG 对应的主 OSD 发送创建 PG 的请求。

函数 OSD::handle_pg_create 用于处理 Monitor 发送的创建 PG 的请求，其消息类型为 MOSDPGCreate 的数据结构：

```
struct MOSDPGCreate : public Message {
  version_t          epoch;
  map<pg_t,pg_create_t> mkpg;     // 要创建的 PG 列表，一次可以创建多个 PG
```

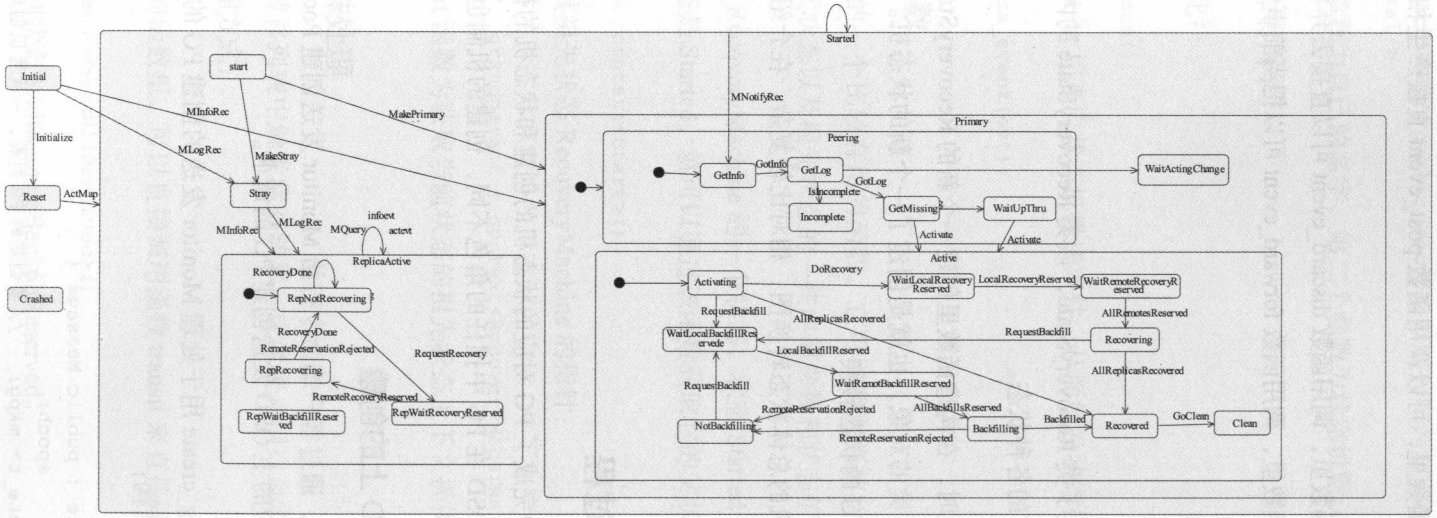


图 10-1　PG 状态机的总体状态转换图

```
  map<pg_t,utime_t> ctimes;          // 对应 PG 的创建时间
}
```

数据结构 pg_create_t 包含了一个 PG 创建相关的参数：

```
struct pg_create_t {
  epoch_t created;        // 创建了 epoch pg
  pg_t parent;            // 如果 parent 不为空 (if != pg_t())，本 PG 不是从 parent 中分
                             裂出来的位
  __s32 split_bits;
}
```

函数 handle_pg_create 的处理过程如下：

1）首先调用函数 require_mon_peer 确保是由 Monitor 发送的创建消息。

2）调用函数 require_same_or_newer_map 检查 epoch 是否一致。如果对方的 epoch 比自己拥有的新，就更新自己的 epoch；否则就直接拒绝该请求。

3）对消息中 mkpg 列表里每一个 PG，开始执行如下创建操作：

a）检查该 PG 的参数 split_bits，如果不为 0，那么就是 PG 的分裂请求，这里不做处理；检查 PG 的 preferred，如果设置了，就跳过，目前不支持；检查确认该 pool 存在；检查本 OSD 是该 PG 的主 OSD；如果参数 up 不等于 acting，说明该 PG 有 temp_pg，至少确定该 PG 存在，直接跳过。

b）调用函数 _have_pg 获取该 PG 对应的类。如果该 PG 已经存在，跳过。

c）调用函数 PG::_create 在本地对象存储中创建相应的 collection。

d）调用函数 _create_lock_pg 初始化 PG。

e）调用函数 pg->handle_create(&rctx) 给新创建 PG 状态机投递事件，PG 的状态发生相应的改变，后面会介绍。

f）所有修改操作都打包在事务 rctx.transaction 中，调用函数 dispatch_context 将事务提交到本地对象存储中。

4）调用函数 maybe_update_heartbeat_peers 来更新 OSD 的心跳列表。

10.3.2　PG 在从 OSD 上的创建

Monitor 并不会给 PG 的从 OSD 发送消息来创建该 PG，而是由该主 OSD 上的 PG 在 Peering 过程中创建。主 OSD 给从 OSD 的 PG 状态机投递事件时，在函数 handle_pg_

peering_evt 中，如果发现该 PG 不存在，才完成创建该 PG。

函数 handle_pg_peering_evt 是处理 Peering 状态机事件的入口。该函数会查找相应的 PG，如果该 PG 不存在，就创建该 PG。该 PG 的状态机进入 RecoveryMachine/Stray 状态。

10.3.3 PG 的加载

当 OSD 重启时，调用函数 OSD::init()，该函数调用 load_pgs 函数加载已经存在的 PG，其处理过程和创建 PG 的过程相似。

10.4 PG 创建后状态机的状态转换

图 10-2 为 PG 总体状态转换图的简化版：状态 Peering、Active、RelicaActive 的内部状态没有添加进去。

通过该图可以了解 PG 的高层状态转换过程，如下所示：

1）当 PG 创建后，同时在该类内部创建了一个属于该 PG 的 RecoveryMachine 类型的状态机，该状态机的初始化状态为默认初始化状态 Initial。

2）在 PG 创建后，调用函数 pg->handle_create(&rctx) 来给状态机投递事件：

```
void PG::handle_create(RecoveryCtx *rctx){
  dout(10) << "handle_create" << dendl;
  Initialize evt;
  recovery_state.handle_event(evt, rctx);
  ActMap evt2;
  recovery_state.handle_event(evt2, rctx);
}
```

由以上代码可知：该函数首先向 RecoveryMachine 投递了 Initialize 类型的事件。由图 10-2 可知，状态机在 RecoveyMachine/Initial 状态接收到 Initialize 类型的事件后直接转移到 Reset 状态。其次，向 RecoveryMachine 投递了 ActMap 事件。

3）状态 Reset 接收到 ActMap 事件，跳转到 Started 状态。

```
boost::statechart::result PG::RecoveryState::Reset::react(const ActMap&){
……
```

```
    return transit< Started >();
  }
```

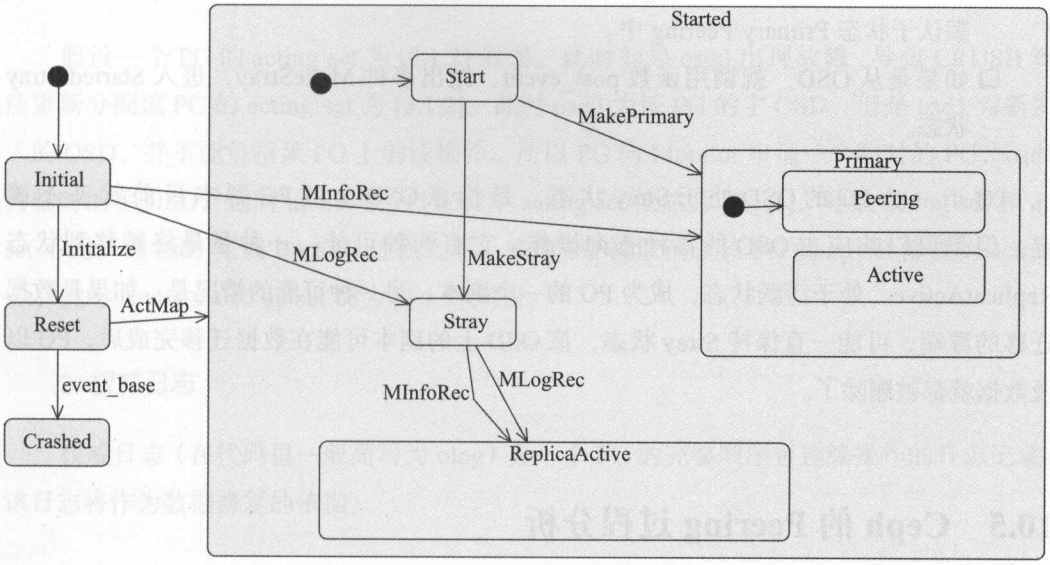


图 10-2　PG 总体状态图的简化版

在自定义的 react 函数里直接调用了 transit 函数跳转到 Started 状态。

4）进入状态 RecoveryMachine/Started 后，就进入 RecoveryMachine/Started 的默认的子状态 RecoveryMachine/Started/Start 中：

```
PG::RecoveryState::Start::Start(my_context ctx)
  : my_base(ctx),
    NamedState(context< RecoveryMachine >().pg->cct, "Start")
{
  context< RecoveryMachine >().log_enter(state_name);
  PG *pg = context< RecoveryMachine >().pg;
  if (pg->is_primary()) {
    dout(1) << "transitioning to Primary" << dendl;
    post_event(MakePrimary());
  } else { //is_stray
    dout(1) << "transitioning to Stray" << dendl;
    post_event(MakeStray());
  }
}
```

由以上代码可知，在 Start 状态的构造函数中，根据本 OSD 在该 PG 中担任的角色不

同分别进行如下处理：

- 如果是主 OSD，就调用函数 post_event，抛出事件 MakePrimary，进入主 OSD 的默认子状态 Primary/Peering 中。
- 如果是从 OSD，就调用函数 post_event，抛出事件 MakeStray，进入 Started/Stray 状态。

对于一个 PG 的 OSD 处于 Stray 状态，是指该 OSD 上的 PG 副本目前状态不确定，但是可以响应主 OSD 的各种查询操作。它有两种可能：一种是最终转移到状态 ReplicatActive，处于活跃状态，成为 PG 的一个副本。另一种可能的情况是：如果是数据迁移的源端，可能一直保持 Stray 状态，该 OSD 上的副本可能在数据迁移完成后，PG 以及数据就都被删除了。

10.5 Ceph 的 Peering 过程分析

在介绍了 statechar 状态机和 PG 的创建过程后，正式开始 Peering 过程介绍。Peering 的过程使一个 PG 内的 OSD 达成一个一致状态。当主从副本达成一个一致的状态后，PG 处于 active 状态，Peering 过程的状态就结束了。但此时该 PG 的三个 OSD 的数据副本上的数据并非完全一致。

PG 在如下两种情况下触发 Peering 过程。

- 当系统初始化时，OSD 重新启动导致 PG 重新加载，或者 PG 新创建时，PG 会发起一次 Peering 的过程。
- 当有 OSD 失效，OSD 的增加或者删除等导致 PG 的 acting set 发生了变化，该 PG 就会重新发起一次 Peering 过程。

10.5.1 基本概念

1. acting set 和 up set

acting set 是一个 PG 对应副本所在的 OSD 列表，该列表是有序的，列表中第一个 OSD 为主 OSD。在通常情况下，up set 和 acting set 列表完全相同。要理解它们的不同之

处，需要理解下面介绍的“临时 PG”概念。

2. 临时 PG

假设一个 PG 的 acting set 为 [0,1,2] 列表。此时如果 osd0 出现故障，导致 CRUSH 算法重新分配该 PG 的 acting set 为 [3,1,2]。此时 osd3 为该 PG 的主 OSD，但是 osd3 为新加入的 OSD，并不能负担该 PG 上的读操作。所以 PG 向 Monitor 申请一个临时的 PG，osd1 为临时的主 OSD，这时 up set 变为 [1,3,2]，acting set 依然为 [3,1,2]，导致 acting set 和 up set 不同。当 osd3 完成 Backfill 过程之后，临时 PG 被取消，该 PG 的 up set 修复为 acting set，此时 acting set 和 up set 都为 [3,1,2] 列表。

3. 权威日志

权威日志（在代码里一般简写为 olog）是一个 PG 的完整顺序且连续操作的日志记录。该日志将作为数据修复的依据。

4. up_thru

引入 up_thru 的概念是为了解决特殊情况：当两个以上的 OSD 处于 down 状态，但是 Monitor 在两次 epoch 中检测到了这种状态，从而导致 Monitor 认为它们是先后宕掉。后宕的 OSD 有可能产生数据的更新，导致需要等待该 OSD 的修复，否则有可能产生数据丢失。

例 10-1　up_thru 处理过程

下图为初始情况：

epoch	处于 up 的 OSD	
1	A	B
2		B
3		
4	A	

过程如下所示：

1）在 epoch1 时，一个 PG 中有 A、B 两个 OSD（两个副本）都处于 up 的状态。

2）在 epoch2 时，Monitor 检测到了 A 处于 down 状态，B 仍然处于 up 状态。由于 Monitor 的检测可能滞后，实际可能有两种情况：

情况 1：此时 B 其实也已经和 A 同时宕了，只是 Monitor 没有检测到。此时 PG 不可能完成 PG 的 Peering 过程，PG 没有新数据写入。

情况 2：此时 B 确实处于 up 状态，由于 B 上保持了完整的数据，PG 可以完成 Peering 过程并处于 active 的状态，可以接受新的数据写操作。

上述两种不同的情况，Monitor 无法区分。

3）在 epoch3 时，Monitor 检测到 B 也宕了。

4）在 epoch4 时，A 恢复了 up 的状态后，该 PG 发起 Peering 过程，该 PG 是否允许完成 Peering 过程处于 active 状态，可以接受读写操作？

- 如果在 epoch2 时，属于情况 1：PG 并没有数据更新，B 上不会新写入数据，A 上的数据保存完整，此时 PG 可以完成 Peering 过程从而处于 active 状态，接受写操作。
- 如果在 epoch2 时，属于情况 2：PG 上有新数据更新到了 osd B，此时 osd A 缺失一些数据，该 PG 不能完成 Peering 过程。

为了使 Monitor 能够区分上述两种情况，引入了 up_thru 的概念，up_thru 记录了每个 OSD 完成 Peering 后的 epoch 值。其初始值设置为 0。

在上述情况 2，PG 如果可以恢复为 active 状态，在 Peering 过程，须向 Monitor 发送消息，Monitor 用数组 up_thru[osd] 来记录该 OSD 完成 Peering 后的 epoch 值。

当引入 up_thru 后，上述例子的处理过程如下：

epoch	处于 up 的 OSD		monitor up_thru
1	A	B	
2		B	up_thru[B]=0
3			
4	A		

情况 1 的处理流程如下：

1）在 epoch1 时，up_thru[B] 为 0，也就是说 B 在 epoch 为 0 时参与完成 Peering。

2）在 epoch2 时，Monitor 检查到 OSD A 处于 down 状态，OSD B 仍处于 up 状态（实际 B 已经处于 down 状态），PG 没有完成 Peering 过程，不会向 Monitor 上报更新 up_thru 的值。

3）epoch3 时，A 和 B 两个 OSD 都宕了。

4）epoch4 时，A 恢复 up 状态，PG 开始 Peering 过程，发现 up_thru[B] 为 0，说明在 epoch 为 2 时没有更新操作，该 PG 可以完成 Peering 过程，PG 处于 active 状态。

情况 2 的处理如下所示：

epoch	处于 up 的 OSD		monitor up_thru
1	A	B	
2		B	up_thru[B]=0
3		B	up_thru[B]=2
4			
5	A		

情况 2 的处理流程如下：

1）在 epoch1 时，up_thru[B] 为 0，也就是说 B 在 epoch 为 0 时参与完成 Peering 过程。

2）在 epoch2 时，Monitor 检查到 OSD A 处于 down 状态，OSD B 还处于 up 状态，该 PG 完成了 Peering 过程，向 Monitor 上报 B 的 up_thru 变为当前 epoch 的值为 2，此时 PG 可接受写操作请求。

3）在 epoch4 时，A 和 B 都宕了，B 的 up_thru 为 2。

4）在 epoch5 时，A 处于 up 状态，开始 Peering 过程，发现 up_thru[B] 为 2，说明在 epoch 为 2 时完成了 Peering，有可能有更新操作，该 PG 需要等待 B 恢复。否则可能丢失 B 上更新的数据。

10.5.2　PG 日志

PG 日志（pg log）为一个 PG 内所有更新操作的记录（下文所指的日志，如不特别指出，都是指 PG 日志）。每个 PG 对应一个 PG 日志，它持久化保存在每个 PG 对应 pgmeta_oid 对象的 omap 属性中。

它有如下的特点：

❑ 记录一个 PG 内所有对象的更新操作元数据信息，并不记录操作的数据。

❑ 是一个完整的日志记录，版本号是顺序的且连续的。

1. pg_log_t

结构体 pg_log_t 在内存中保存了该 PG 的所有操作日志，以及相关的控制结构。

```
struct pg_log_t {
eversion_t head;     // 日志的头，记录最新的日志记录
eversion_t tail;     // 日志的尾，记录最旧的日志记录

eversion_t can_rollback_to;     // 用于 EC，指示本地可以回滚的版本，可回滚的版本都大于版本 can_rollback_to 的值
eversion_t rollback_info_trimmed_to;
                // 在 EC 的实现中，本地保留了不同版本的数据。本数据段指示本 PG 里可以删除掉的对象版本
list<pg_log_entry_t> log;  // 所有日志的列表
......
}
```

需要注意的是，PG 日志的记录是以整个 PG 为单位，包括该 PG 内所有对象的修改记录。

2. pg_log_entry_t

结构体 pg_log_entry_t 记录了 PG 日志的单条记录，其数据结构如下：

```
struct pg_log_entry_t {
  __s32 op;                    // 操作的类型
  hobject_t soid;              // 操作的对象
  eversion_t version,          // 本次操作的版本
  prior_version,               // 前一个操作的版本
  reverting_to;                // 本次操作回退的版本（仅用于回滚操作）

  ObjectModDesc mod_desc;      // 用于保存本地回滚的一些信息，用于 EC 模式下的回滚操作

  bufferlist snaps;            // 克隆操作用于记录当前对象的 snap 列表
  osd_reqid_t reqid;           // 请求唯一标识（called + tid）
  vector<pair<osd_reqid_t, version_t> > extra_reqids;
  version_t user_version;      // 用户的版本
  utime_t     mtime;           // 这是用户本地时间
  ......
}
```

3. IndexedLog

类 IndexedLog 继承了类 pg_log_t，在其基础上添加了根据一个对象来检索日志的功能，以及其他相关的功能。

4. 日志的写入

函数 PG::add_log_entry 添加 pg_log_entry_t 条目到 PG 日志中。同时更新了 info.last_complete 和 info.last_update 字段。

PGLog::write_log 函数将日志写到对应的 pgmeta_oid 对象的 kv 存储中。在这里并没有直接写入磁盘，而是先把日志的修改添加到 ObjectStore::Transaction 类型的事务中，与数据操作组成一个事务整体提交磁盘。这样可以保证数据操作、日志更新及其 pg info 信息的更新都在一个事务中，都以原子方式提交到磁盘上。

5. 日志的 trim 操作

函数 trim 用来删除不需要的旧日志。当日志的条目数大于 min_log_entries 时，需要进行 trim 操作。

```
void PGLog::trim(LogEntryHandler *handler,
eversion_t trim_to, pg_info_t &info)
```

6. 合并权威日志

函数 merge_log 用于把本地日志和权威日志合并：

```
void PGLog::merge_log(ObjectStore::Transaction& t,
                      pg_info_t &oinfo, pg_log_t &olog,
                      pg_shard_t fromosd,
                      pg_info_t &info, LogEntryHandler *rollbacker,
                      bool &dirty_info, bool &dirty_big_info)
```

其处理过程如下：

1）本地日志和权威日志没有重叠的部分：在这种情况下就无法依据日志来修复，只能通过 Backfill 过程来完成修复。所以先确保权威日志和本地日志有重叠的部分：

```
assert(log.head >= olog.tail && olog.head >= log.tail);
```

2）本地日志和权威日志有重叠部分的处理：

- 如果 olog.tail 小于 log.tail，也就是权威日志的尾部比本地日志长。在这种情况下，只要把日志多出的部分添加到本地日志即可，它不影响 missing 对象集合。
- 本地日志的头部比权威日志的头部长，说明有多出来的 divergent 日志，调用函数 rewind_divergent_log 去处理。
- 本地日志的头部比权威日志的头部短，说明有缺失的日志，其处理过程为：把缺失的日志添加到本地日志中，记录 missing 的对象，并删除多出来的日志记录。

下面举例说明函数 merge_log 的不同处理情况。

例 10-2 函数 merge_log 应用举例

情况 1：权威日志的尾部版本比本地日志的尾部小，如下所示：

log（本地日志）			obj10(1,6) modify log_tail	obj11(1,7) modify	obj13(1,8) modify log_head
olog（权威日志）	obj3(1,4) modify log_tail	obj4(1,5) modify	obj10(1,6) modify	obj11(1,7) modify	obj13(1,8) modify log_head

本地 log 的 log_tail 为 obj10（1,6），权威日志 olog 的 log_tail 为 obj3（1,4）。

日志合并的处理方式如下所示：

log（本地日志）	obj3 (1,4) modify log_tail	obj4(1,5) modify	obj10(1,6) modify ~~log_tail~~	obj11(1,7) modify	obj13(1,8) modify
olog（权威日志）	obj3 (1,4) modify log_tail	obj4(1,5) modify	obj10(1,6) modify	obj11(1,7) modify	obj13(1,8) modify

把日志记录 obj3（1,4）、obj4（1,5）添加到本地日志中，修改 info.log_tail 和 log.tail 指针即可。

情况 2：本地日志的头部版本比权威日志长，如下所示：

olog	obj10(1,6) modify	obj11(1,7) modify	obj13(1,8) modify log_head			
log	obj10(1,6) modify	obj11(1,7) modify	obj13(1,8) modify	obj13(1,9) modify	obj11(1,10) modify	obj10(1,11) delete log_head

权威日志的 log_head 为 obj13（1,8），而本地日志的 log_head 为 obj10(1,11)。本地日志的 log_head 版本大于权威日志的 log_head 版本，调用函数 rewind_divergent_log 来处理本地有分歧的日志。

在本例的具体处理过程为：把对象 obj10、obj11、obj13 加入 missing 列表中用于修复。最后删除多余的日志，如下所示：

olog	obj10(1,6) modify	obj11(1,7) modify	obj13(1,8) modify log_head			
log	obj10(1,6) modify	obj11(1,7) modify	obj13(1,8) modify log_head	~~obj13(1,9)~~ ~~modify~~	~~obj11(1,10)~~ ~~modify~~	~~obj10(1,11)~~ ~~delete~~ ~~log_head~~
missing object：obj10（1,6）obj11(1,7) obj13(1,8)						

本例比较简单，函数 rewind_divergent_log 会处理比较复杂的一些情况，后面会介绍到。

情况 3：本地日志的头部版本比权威日志的头部短，如下所示：

olog	obj10(1,6) modify	obj11(1,7) modify	obj13(1,8) modify	obj13(1,9) modify	obj11(1,10) modify	obj10(1,11) delete log_head
log	obj10(1,6) modify	obj11(1,7) modify	obj13(1,8) modify log_head			

权威日志的 log_head 为 obj10（1,11），而本地日志的 log_head 为 obj13(1,8)，即本地日志的 log_head 版本小于权威日志的 log_head 版本。

其处理方式如下：把本地日志缺失的日志添加到本地，并计算本地缺失的对象。最后把缺失的对象添加到 missing object 列表中用于后续的修复，处理结果如下所示：

olog	obj10(1,6) modify	obj11(1,7) modify	obj13(1,8) modify	obj13(1,9) modify	obj11(1,10) modify	obj10(1,11) delete log_head
log	obj10(1,6) modify	obj11(1,7) modify	obj13(1,8) modify ~~log_head~~	obj13(1,9) modify	obj11(1,10) modify	obj10(1,11) delete log_head
missing object			obj13(1,9)　obj11(1,10)			

7. 处理副本日志

函数 proc_replica_log 用于处理其他副本节点发过来的和权威日志有分叉（divergent）的日志。其关键在于计算 missing 的对象列表，也就是需要修复的对象，如下所示：

```
void PGLog::proc_replica_log(ObjectStore::Transaction& t,
                             pg_info_t &oinfo, const pg_log_t &olog,
                             pg_missing_t& omissing,
                             pg_shard_t from) const
```

其参数都为远程节点的信息：

- oinfo：远程节点的 pg_info_t。
- olog：远程节点的日志。
- omissing：远程节点的 missing 信息，为输出信息。
- from：来自远程节点。

函数 proc_replica_log 的具体处理过程如下：

1）如果日志不重叠，就无法通过日志来修复，需要进行 Backfill 过程，直接返回。

2）如果日志的 head 相同，说明没有分歧日志（divergent log），直接返回。

3）下面处理的都是这种情况：日志有重叠并且日志的 head 不相同，需要处理分歧的日志：

- 计算第一条分歧日志 first_non_divergent，从本地日志后往前查找小于等于 olog.head 的日志记录。
- 版本 lu 为分歧日志的边界。如果 first_non_divergent 没有找到，或者小于权威日志的 log_tail，那么 lu 就设置为 log_tail，否则就设置为 first_non_divergent 日志记录的版本。
- 把所有的分歧日志都添加到 divergent 队列里。
- 构建一个 IndexedLog 的对象 folog，把所有没有分歧的日志添加到 folog 里。
- 调用函数 _merge_divergent_entries 处理分歧日志。
- 更新 oinfo 的 last_update 为 lu 版本。
- 如果有对象 missing，就设置 last_complete 为小于 first_missing 的版本。

函数 _merge_divergent_entries 处理所有的分歧日志，首先把所有分歧日志的对象按照对象分类，然后分别调用函数 _merge_object_divergent_entries 对每个分歧日志的对象进行处理。

函数 _merge_object_divergent_entries 用于处理单个对象的 divergent 日志，其处理过程如下：

1）首先进行比较，如果处理的对象 hoid 大于 info.last_backfill，说明该对象本来就不存在，没有必要修复。

这种情况一般发生在如下情景：该 PG 在上一次 Peering 操作成功后，PG 还没有处理 clean 状态，正在 Backfill 过程中，就再次触发了 Peering 的过程。info.last_backfill 为上次最后一个修复的对象。

在本 PG 完成 Peering 后就开始修复，先完成 Recovery 操作，然后会继续完成上次的 Backfill 操作，所以没有必要在这里检查来修复。

2）通过该对象的日志记录来检查版本是否一致。首先确保是同一个对象，本次日志记录的版本 prior_version 等于上一条日志记录的 version 值。

3）版本 first_divergent_update 为该对象的日志记录中第一个产生分歧的版本；版本 last_divergent_update 为最后一个产生分歧的版本；版本 prior_version 为第一个分歧产生的前一个版本，也就是应该存在的对象版本。布尔变量 object_not_in_store 用来标记该对象不缺失，且第一条分歧日志操作是删除操作。处理分歧日志的五种情况如下所示：

情况 1：在没有分歧的日志里查找到该对象，但是已存在的对象的版本大于第一个分歧对象的版本。这种情况的出现，是由于在 merge_log 中产生权威日志时的日志更新，相应的处理已经做了，这里不做任何处理。

情况 2：如果 prior_version 为 eversion_t()，为对象的 create 操作或者是 clone 操作，那么这个对象就不需要修复。如果已经在 missing 记录中，就删除该 missing 记录。

情况 3：如果该对象已经处于 missing 列表中，如下进行处理：

- 如果日志记录显示当前已经拥有的该对象版本 have 等于 prior_version，说明对象不缺失，不需要修复，删除 missing 中的记录。

- 否则，修改需要修复的版本 need 为 prior_version；如果 prior_version 小于等于 info.log_tail 时，这是不合理的，设置 new_divergent_prior 用于后续处理。

情况 4：如果该对象的所有版本都可以回滚，直接通过本地回滚操作就可以修复，不需要加入 missing 列表来修复。

情况 5：如果不是所有的对象版本都可以回滚，删除相关的版本，把 prior_version 加入 missing 记录中用于修复。

10.5.3 Peering 的状态转换图

由 10.4 节分析可知，主 OSD 上 PG 对应的状态机 RecoveryMachine 目前已经处于 Started/Primary/Peering 状态。从 OSD 上的 PG 对应的 RecoveryMachine 处于 Started/Stray 状态。本节总体介绍 Peering 过程的状态转换。

如图 10-3 所示为 Peering 状态转换图，其过程如下：

1）当进入 Primary/Peering 状态后，就进入默认子状态 GetInfo 中。
2）状态 GetInfo 接收事件 GotInfo 后，转移到 GetLog 状态中。
3）如果状态 GetLog 接收到 IsIncomplete 事件后，跳转到 Incomplete 状态。
4）状态 GetLog 接收到事件 GotLog 后，就转入 GetMissing 状态。
5）状态 GetMissing 接收到事件 Activate 事件，转入状态 active 状态。

由上述 Peering 的状态转换过程可知，Peering 过程基本分为如下三个步骤：

步骤 1：GetInfo：PG 的主 OSD 通过发送消息获取所有从 OSD 的 pg_info 信息。

步骤 2：GetLog：根据各个副本获取的 pg_info 信息的比较，选择一个拥有权威日志的 OSD（auth_log_shard）。如果主 OSD 不是拥有权威日志的 OSD，就从该 OSD 上拉取权威日志。主 OSD 完成拉取权威日志后也就拥有了权威日志。

步骤 3：GetMissing：主 OSD 拉取其他从 OSD 的 PG 日志（或者部分获取，或者全部获取 FULL_LOG）。通过与本地权威日志的对比，来计算该 OSD 上缺失的 object 信息，作为后续 Recovery 操作过程的依据。

最后通过 Active 操作激活主 OSD，并发送 notify 通知消息，激活相应的从 OSD。

下面介绍这三个主要步骤。

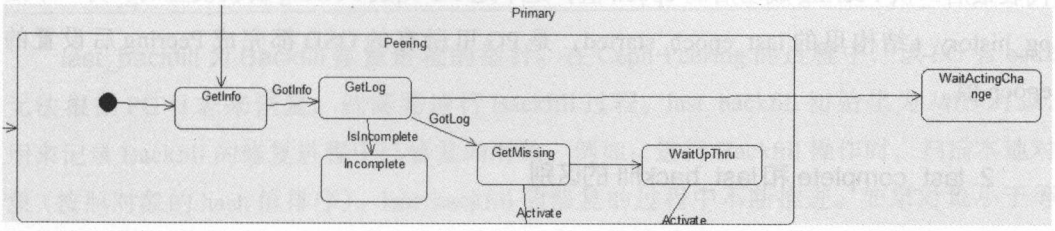


图 10-3　Peering 状态转换图

10.5.4　pg_info 数据结构

数据结构 pg_info_t 保存了 PG 在 OSD 上的一些描述信息。该数据结构在 Peering 的整个过程，以及后续的数据修复中都发挥了重要作用，理解该数据结构的各个关节字段的含义可以更好地理解相关的过程。pg_info_t 数据结构如下：

```
struct pg_info_t{
  spg_t pgid;                          //PG 的 id
eversion_t last_update;                //PG 最后一次更新的版本
eversion_t last_complete;
  epoch_t last_epoch_started;
  version_t last_user_version;         // 最后更新的用户版本号，用于分层存。
eversion_t log_tail;                   // 日志的尾部版本
  hobject_t last_backfill;             // 上一次 Backfill 操作的对象指针。如果该 OSD 的
Backfill 操作没有完成，那么 last_backfill 和 last_complete 之间的对象就丢失。

interval_set<snapid_t> purged_snaps;   //PG 要删除的 snap 集合

  pg_stat_t stats;                      //PG 的统计信息
  pg_history_t history;                 //PG 的历史信息
  pg_hit_set_history_t hit_set;         // 这是 Cache Tier 用的 hit_set
}
```

结构 pg_history_t 保存了 PG 的一些历史信息：

```
struct pg_history_t {
epoch_t epoch_created;              //PG 创建的 epoch 值
  epoch_t last_epoch_started;
epoch_t last_epoch_clean;           //PG 处于 clean 状态时的 epoch 值
  ......
}
```

1. last_epoch_started 介绍

last_epoch_started 字段有两个地方出现，一个是 pg_info 结构里的 last_epoch_started，代表最后一次 Peering 成功后的 epoch 值，是本地 PG 完成 Peering 后就设置的。另一个是 pg_history_t 结构里的 last_epoch_started，是 PG 里所有的 OSD 都完成 Peering 后设置的 epoch 值。

2. last_complete 和 last_backfill 的区别

在这里特别指出 last_update 和 last_complete、last_backfill 之间的区别。下面通过一个例子来讲解，同时也可以大概了解 PG 数据恢复的流程。在数据恢复过程中先进行 Recovery 过程，再进行 Backfill 过程（可以参考第 11 章的详细介绍）。

情况 1： 在 PG 处于 clean 状态时，last_complete 就等于 last_update 的值，并且等于 PG 日志中的 head 版本。它们都同步更新，此时没有区别。last_bacfill 设置为 MAX 值。例如：下面的 PG 日志里有三条日志记录。此时 last_update 和 last_complete 以及 pg_log.head 都指向版本（1,2）。由于没有缺失的对象，不需要恢复，last_backfill 设置为 MAX 值。示例如下所示：

obj1（1,0）	obj2（1,1）	obj3（1,2） last_update last_complete pg_log.head		

情况 2： 当该 osd1 发生异常之后，过一段时间后又重新恢复，当完成了 Peering 状态后的情况。此时该 PG 可以继续接受更新操作。例如：下面的灰色字体的日志记录为该 osd1 崩溃期间缺失的日志，obj7 为新的写入的操作日志记录。last_update 指向最新的更新版本（1,7），last_complete 依然指向版本（1,2）。即 last_update 指的是最新的版本，last_complete 指的是上次的更新版本。过程如下：

obj1（1,0）	obj2（1,1）	obj3（1,2） last_complete	obj4(1,3)	obj3(1,4)	obj5(1,5)	obj6(1,6)	obj7(1,7) last_update pg_log.head

last_complete 为 Recovery 修复进程完成的指针。当该 PG 开始进行 Recovery 工作时，last_complete 指针随着 Recovery 过程推进，它指向完成修复的版本。例如：当

Recovery 完成后 last_complete 指针指向最后一个修复的对象的版本（1,6），如下所示：

obj1（1,0）	obj2（1,1）	obj3（1,2）	obj4(1,3)	obj3(1,4)	obj5(1,5)	obj6(1,6) last_complete	obj7(1,7) last_update pg_log.head

last_backfill 为 Backfill 修复进程的指针。在 Ceph Peering 的过程中，该 PG 有 osd2 无法根据 PG 日志来恢复，就需要进行 Backfill 过程。last_backfill 初始化为 MIN 对象，用来记录 Backfill 的修复进程中已修复的对象。例如：进行 Backfill 操作时，扫描本地对象（按照对象的 hash 值排序）。last_backfill 随修复的过程中不断推进。如果对象小于等于 last_backfill，就是已经修复完成的对象。如果对象大于 last_backfill 且对象的版本小于 last_complete，就是处于缺失还没有修复的对象。过程如下所示：

Backfill 对象的列表	MIN	obj1（1,0）	ojb2（1,1）	obj3（1,4）	obj4（1,3）	obj5（1,5）	obj6（1,6）
	last_backfill						last_complete

当恢复完成之后，last_backfill 设置为 MAX 值，表明恢复完成，设置 last_complete 等于 last_update 的值。

10.5.5　GetInfo

GetInfo 过程获取该 PG 在其他 OSD 上的结构图 pg_info_t 信息（也称 pg_info 信息）。这里的其他 OSD 包括当前 PG 的活跃 OSD，以及 past interval 期间该 PG 所有处于 up 状态的 OSD。

由 10.5.4 节的介绍可知，当 PG 进入 Primary/Peering 状态后，就进入默认的子状态 GetInfo 里。其主要流程在构造函数里完成：

```
PG::RecoveryState::GetInfo::GetInfo(my_context ctx)
  : my_base(ctx),
  NamedState(context< RecoveryMachine >().pg->cct, "Started/Primary/Peering/
GetInfo")
  {
  ......
}
```

在构造函数 GetInfo 里，完成了核心的功能，实现过程如下：

1）调用函数 generate_past_intervals 计算 past intervals 的值：

```
pg->generate_past_intervals();
```

2）调用函数 build_prior 构造获取 pg_info_t 信息的 OSD 列表：

```
pg->build_prior(prior_set);
```

3）调用函数 get_infos 给参与的 OSD 发送获取请求：

```
get_infos();
```

由上述可知，GetInfo 过程基本分三个步骤：计算 past_interval 的过程；通过调用函数 build_prior 来计算要获取 pg_info 信息的 OSD 列表；最后调用函数 get_infos 给相关的 OSD 发送消息来获取 pg_info 信息，并处理接收到的 Ack 应答。

1. 计算 past_interval

函数 past_interval 是 epoch 的一个序列。在该序列内一个 PG 的 acting set 和 up set 不会变化。current_inteval 是一个特殊的 past_interval，它是当前最新的一个没有变化的序列。示例如下：

说明如下：

1）Ceph 系统当前的 epoch 值为 20，PG1.0 的 acting set 和 up set 都为列表 [0,1,2]。

2）osd3 失效导致了 osd map 变化，epoch 变为 21。

3）osd5 失效导致了 osd map 变化，epoch 变为 22。

4）osd6 失效导致了 osd map 变化，epoch 变为 23。

上述三次 epoch 的变化都不会改变 PG1.0 的 acting set 和 up set。

epoch	20	21	22	23	24	25	26
失效 OSD		osd3	osd5	osd6	osd2	osd12	osd13
PG 1.0	[0,1,2]	[0,1,2]	[0,1,2]	[0,1,2]	[0,1,8]	[0,1,8]	[0,1,8]
	[0,1,2]	[0,1,2]	[0,1,2]	[0,1,2]	[0,1,8]	[0,1,8]	[0,1,8]
					last_epoch_started	last_epoch_clean	
	past_interval				current_interval		

5）osd2 失效导致了 osd map 变化，epoch 变为 24；此时导致 PG1.0 的 acting set 和 up set 变为 [0,1,8]，若此时 Peering 过程成功完成，则 last_epoch_started 为 24。

6）osd12 失效导致了 osd map 变化，epoch 变为 25，此时如果 PG1.0 完成了 Recovery 操作，处于 clean 状态，last_epoch_clean 就为 25。

7）osd13 失效导致了 osd map 变化，epoch 变为 26。

epoch 序列 [20,21,22,23] 就为 PG1.0 的一个 past interval，epoch 序列 [24,25,26] 就为 PG1.0 的 current interval。

数据结构 pg_interval_t 用于保存 past_interval 的信息：

```
struct pg_interval_t {
vector<int32_t> up, acting;    // 在本 interval 阶段 PG 处于 up 和 acting 状态的 OSD
  epoch_t first, last;         // 起始和结束的 epoch
  bool maybe_went_rw;          // 在这个阶段是否有数据读写操作
  int32_t primary;             // 主 OSD
  int32_t up_primary;          //up 状态的主 OSD
};
```

上例中，past_interval 对象的 p 值为：

```
p =  {
    up = [0,1,2],
    acting = [0,1,2],
    first = 20,
    last = 23,
    maybe_went_rw = true,
    primary = 0,
    up_primay = 0,
}
```

函数 generate_past_intervals 用于计算 past_intervals 的值，计算的结果保存在 PG 中 past_intervals 的 map 结构里，map 的 key 值为 first epoch 的值：

```
map<epoch_t,pg_interval_t> past_intervals;
```

具体计算过程如下：

1）调用函数 _calc_past_interval_range 推测需要计算的 past_interval 的起始 epoch 值（start）和结束 epoch 值（end）。如果返回 false，说明不需要计算 past_interval，所有的 past_interval 已经计算好了。

2）从 start 到 end 开始计算 past_interval。过程为调用函数 check_new_interval 比较两次 epoch 对应的 osd map 的变化。如果检查是一个新值，就创建一个新的 past_interval 对象。

```
bool PG::_calc_past_interval_range(epoch_t *start, epoch_t *end, epoch_t oldest_map)
```

函数 _calc_past_interval_range 用于计算 past_interval 的范围。参数 oldest_map 为 OSD 的 superblock 里保存的最老 osd map，输出为 start 和 end，分别为需要计算的 past_interval 的 start 和 end 值。具体实现过程如下。

计算 end 值如下所示：

1）变量 end 为当前 osd map 的 epoch 值，如果 info.history.same_interval_since 不为空，就设置为该值。该值表示和当前的 osd map 的 epoch 值在同一个 interval 中。

```
if (info.history.same_interval_since) {
  *end = info.history.same_interval_since;
} else {
  // 当前 PG 可能是新引入的，计算整个 range 期间 interval
  *end = osdmap_ref->get_epoch();
}
```

2）查看 past_intervals 里已经计算的 past_interval 的第一个 epoch，如果已经比 info.history.last_epoch_clean 小，就不用计算了，直接返回 false 值。否则设置 end 为其 first 值。

```
*end = past_intervals.begin()->first;
```

计算 start 值如下所示：

1）start 设置为 info.history.last_epoch_clean，从最后一次 last_epoch_clean 算起。

2）当 PG 为新建时，从 info.history.epoch_created 开始计算。

3）oldest_map 值为保存的最早 osd map 的值，如果 start 小于这个值，相关的 osd map 信息缺失，所以无法计算。

所以将 start 设置为三者的最大值：

```
*start = MAX(MAX(info.history.epoch_created,
               info.history.last_epoch_clean),
             oldest_map);
```

下面举例说明计算 past_interval 的过程。

例 10-3　past_interval 计算示例

4	5	6	7	8	9	10	11	12	13	14	15	16
past_interval 1					past_interval 2			past_interval 3		current_interval		

如上表所示：一个 PG 有 4 个 interval。past_interval 1，开始 epoch 为 4，结束的 epoch 为 8；past_interval 2 的 epoch 区间为（9,11）；past_interval 3 的区间为（12,13）；current_interval 的区间为（14,16）。最新的 epoch 为 16，info.history.same_interval_since 为 14，意指是从 epoch14 开始，之后的 epoch 值和当前的 epoch 值在同一个 interval 内。info.history.last_epoch_clean 为 8，就是说在 epoch 值为 8 时，该 PG 处于 clean 状态。

计算 start 和 end 的方法如下：

1）start 的值设置为 info.history.last_epoch_clean 值，其值为 8。

2）end 值从 14 开始计算，检查当前已经计算好的 past_intervals 的值。past_interval 的计算是从后往前计算。如果第一个 past interval 的 first 小于等于 8，也就是 past_interval 1 已经计算过了，那么后面的 past_interval 2 和 past_interval 3 都已经计算过，就直接退出。否则就继续查找没有计算过的 past_interval 的值。

2. 构建 OSD 列表

函数 build_prior 根据 past_intervals 来计算 probe_targets 列表，也就是要去获取 pg_info 的 OSD 列表。具体实现为：首先重新构造一个 PriorSet 对象，在 PriorSet 的构造函数中完成下列操作：

1）把当前 PG 的 acting set 和 up set 中的 OSD 加入到 probe 列表中。

2）检查每个 past_intervals 阶段：

a）如果 interval.last 小于 info.history.last_epoch_started，这种情况下 past_interval 就没有意义，直接跳过。

b）如果该 interval 的 act 为空，就跳过。

c）如果该 interval 没有 rw 操作，就跳过。

d）对于当前 interval 的每一个处于 acting 状态的 OSD 进行检查：

- 如果该 OSD 当前处于 up 状态，就加入到 up_now 列表中。同时加入到 probe

列表中，用于获取权威日志以及后续数据恢复。

- 如果该 OSD 当前不是 up 状态，但是在该 past_interval 期间还处于 up 状态，就加入 up_now 列表中。
- 否则加入 down 列表，该列表保存所有宕了的 OSD。
- 如果当前 interval 确实有宕的 OSD，就调用函数 pcontdec，也就是 PG 的 IsPGRecoverablePredicate 函数。该函数判断该 PG 在该 interval 期间是否可恢复。如果无法恢复，直接设置 pg_down 为 true 值。

注意 这里特别强调的是，要确保每个 interval 期间都可以进行修复。函数 IsPGRecoverablePredicate 实际上是一个类的运算符重载。对于不同类型的 PG 有不同的实现。对于 ReplicatedPG 对应的实现类为 RPCRecPred，其至少保证有一个处于 up 状态的 OSD；对应 ErasureCode（n+m）类型的 PG，至少有 n 个处于 up 状态的 OSD。

3）如果 prior.pg_down 设置为 true，就直接设置 PG 为 PG_STATE_DOWN 状态。

4）检查是否需要 need_up_thru 设置。

5）用 prior_set->probe 设置 probe_targets 列表。

3. 获取 pg_info 信息

在上述过程中计算出了 PG 在 past interval 期间以及当前处于 up 状态的 OSD 列表，下面就发送请求给 OSD 来获取 pg_info 信息：

```
void PG::RecoveryState::GetInfo::get_infos()
```

函数 get_infos 向 prior_set 的 probe 集合中的每个 OSD 发送 pg_query_t::INFO 消息，来获取 PG 在该 OSD 上的 pg_info 信息。发送消息的过程调用 RecoveryMachine 类的 send_query 函数来进行：

```
context< RecoveryMachine >().send_query(
  peer,
  pg_query_t(pg_query_t::INFO,
    it->shard, pg->pg_whoami.shard,
    pg->info.history,
    pg->get_osdmap()->get_epoch())
    );
  peer_info_requested.insert(peer);
```

```
pg->blocked_by.insert(peer.osd)
```

数据结构 pg_notify_t 定义了获取 pg_info 的 ACK 信息：

```
struct pg_notify_t {
  epoch_t query_epoch;      // 查询时请求消息的 epoch
  epoch_t epoch_sent;       // 发送时应对消息的 epoch
  pg_info_t info;           // pg_info 的信息
  shard_id_t to;            // 目标 OSD
  shard_id_t from;          // 源 OSD
};
```

在主 OSD 收到 pg_info 的 ACK 消息后封装成 MNotifyRec 事件发送给该 PG 对应的状态机。在下列的事件处理函数中来处理 MNotifyRec 事件。

```
boost::statechart::result PG::RecoveryState::GetInfo::react(const MNotifyRec&
infoevt)
```

具体处理过程如下：

1）首先从 peer_info_requested 里删除该 peer，同时从 blocked_by 队列里删除。

2）调用函数 PG::proc_replica_info 来处理副本的 pg_info 消息：

a）首先检查如果该 OSD 的 pg_info 信息，如果已经存在，并且 last_update 参数相同，则说明已经处理过，返回 false 值。否则保存该 pg_info 的值。

b）调用函数 has_been_up_since 检查该 OSD 在 send_epoch 时已经处于 up 状态。

c）确保自己是主 OSD，把该 OSD 的 pg_info 信息保存到 peer_info 数组，并加入 might_have_unfound 数组里。该数组里的 OSD 用于后续的数据恢复。

d）调用函数 unreg_next_scrub 使该 PG 不在 scrub 操作的队列中。

e）调用 info.history.merge 函数处理从 OSD 发过来的 pg_info 信息。处理方法是：更新为最新的字段，设置 dirty_info 为 true 值。

f）调用函数 reg_next_scrub 注册 PG 下一次的 scrub 的时间。

g）如果该 OSD 既不在 up 数组中也不在 acting 数组中，那就加入 stray_set 列表中。当 PG 处于 clean 状态时，就会调用 purge_strays 函数删除 stray 状态的 PG 及其上的对象数据。

h）如果是一个新的 OSD，就调用函数 update_heartbeat_peers 更新需要 heartbeat 的 OSD 列表。

3）在变量 old_start 里保存了调用 proc_replica_info 前主 OSD 的 pg->info.history.last_epoch_started，如果该 epoch 值小于合并后的值，说明该值被更新了，从 OSD 上的 epoch 值比较新，需要进行如下操作：

a）调用 pg->build_prior 重新构建 prior_set 对象。

b）从 peer_info_requested 队列中去掉上次构建的 prior_set 中存在的 OSD，这里最新构建上次不存在的 OSD 列表。

c）调用 get_infos 函数重新发送查询 peer_info 请求。

4）调用 pg->apply_peer_features 更新相关的 features 值。

5）当 peer_info_requested 队列为空，并且 prior_set 不处于 pg_down 的状态时，说明收到所有 OSD 的 peer_info 并处理完成。

6）最后检查 past_interval 阶段至少有一个 OSD 处于 up 状态且非 incomplete 状态；否则该 PG 无法恢复，标记状态为 PG_STATE_DOWN 并直接返回。

7）最后完成处理，调用函数 post_event(GotInfo()) 抛出 GetInfo 事件进入状态机的下一个状态。

在 GetInfo 状态里直接定义了当前状态接收到 GotInfo 事件后，直接跳转到下一个状态 GetLog 里：

```
struct GetInfo : boost::statechart::state< GetInfo, Peering >, NamedState {
  typedef boost::mpl::list <
    boost::statechart::custom_reaction< QueryState >,
    boost::statechart::transition< GotInfo, GetLog >,
    boost::statechart::custom_reaction< MNotifyRec >
  > reactions;
}
```

10.5.6 GetLog

当 PG 的主 OSD 获取到所有从 OSD（以及 past interval 期间的所有参与该 PG 且目前仍处于 active 状态的 OSD）的 pg_info 信息后，就跳转到 GetLog 状态。

```
PG::RecoveryState::GetLog::GetLog(my_context ctx)
```

然后在 GetLog 的构造函数里做相应的处理，其具体处理过程分析如下：

1）调用函数 pg->choose_acting(auth_log_shard) 选出具有权威日志的 OSD，并计算出 acting_backfill 和 backfill_targets 两个 OSD 列表。输出保存在 auth_log_shard 里。

2）如果选择失败并且 want_acting 不为空，就抛出 NeedActingChange 事件，状态机转移到 Primary/WaitActingChang 状态，等待申请临时 PG 返回结果。如果 want_acting 为空，就抛出 IsIncomplete 事件，PG 的状态机转移到 Primay/Peering/Incomplete 状态。表明失败，PG 就处于 Incomplete 状态。

3）如果 auth_log_shard 等于 pg->pg_whoami 的值，也就是选出的拥有权威日志的 OSD 为当前主 OSD，直接抛出事件 GotLog() 完成 GetLog 过程。

4）如果 pg->info.last_update 小于权威 OSD 的 log_tail，也就是本 OSD 的日志和权威日志不重叠，那么本 OSD 无法恢复，抛出 IsIncomplete 事件。经过函数 choose_acting 的选择后，主 OSD 必须是可恢复的。如果主 OSD 不可恢复，必须申请一个临时 PG，选择拥有权威日志的 OSD 为临时主 OSD。

5）如果自己不是权威日志的 OSD，则需要去拥有权威日志的 OSD 上去拉取权威日志，并与本地合并。

1. choose_acting

函数 choose_acting 用来计算 PG 的 acting_backfill 和 backfill_targets 两个 OSD 列表。acting_backfill 保存了当前 PG 的 acting 列表，包括需要进行 Backfill 操作的 OSD 列表；backfill_targets 列表保存了需要进行 Backfill 的 OSD 列表。其处理过程如下：

1）首先调用函数 find_best_info 来选举出一个拥有权威日志的 OSD，保存在变量 auth_log_shard 里。

2）如果没有选举出拥有权威日志的 OSD，则进入如下流程：

a）如果 up 不等于 acting，申请临时 PG，返回 false 值。

b）否则确保 want_acting 列表为空，返回 false 值。

3）计算是否是 compat_mode 模式，检查是，如果所有的 OSD 都支持纠删码，就设置 compat_mode 值为 true。

4）根据 PG 的不同类型，调用不同的函数，对应 ReplicatedPG 调用函数 calc_replicated_acting 来计算 PG 的需要列表：

```
set<pg_shard_t> want_backfill, want_acting_backfill;
  //want_backfill 为该 PG 需要进行 Backfill 的 pg_shard
  //want_acting_backfill 包括进行 acting 和 Backfill 的 pg_shard
pg_shard_t want_primary; // 主 OSD
vector<int> want;          // 在 compat_mode 模式下，和 want_acting_backfill 相同
```

5）下面就是对 PG 做的一些检查：

a）计算 num_want_acting 数量，检查如果小于 min_size，进行如下操作：

- 如果对于 EC，清空 want_acting，返回 false 值。
- 对于 ReplicatePG，如果该 PG 不允许小于 min_size 的恢复，清空 want_acting，返回 false 值。

b）调用 IsPGRecoverablePredicate 来判断 PG 现有的 OSD 列表是否可以恢复，如果不能恢复，清空 want_acting，返回 false 值。

6）检查如果 want 不等于 acting，设置 want_acting 为 want：

a）如果 wang_acting 等于 up，申请 empty 为 pg_temp 的 OSD 列表。

b）否则申请 want 为 pg_temp 的 OSD 列表。

7）最后设置 PG 的 actingbackfill 为 want_acting_backfill，设置 backfill_targets 为 want_backfill，并检查 backfill_targets 里的 pg_shard 应该不在 stray_set 里面。

8）最终返回 true 值。

下面举例说明需要申请 pg_temp 的场景：

1）当前 PG1.0，其 acting 列表和 up 列表都为 [0,1,2]，PG 处于 clean 状态。

2）此时，osd0 崩溃，导致该 PG 经过 CRUSH 算法重新获得 acting 和 up 列表都为 [3,1,2]。

3）选择出拥有权威日志的 osd 为 1，经过 calc_replicated_acting 算法，want 列表为 [1,3,2]，acting_backfill 为 [1,3,2], want_backfill 为 [3]。特别注意 want 列表第一个为主 OSD，如果 up_primay 无法恢复，就选择权威日志的 OSD 为主 OSD。

4）want[1,3,2] 不等于 acting[3,1,2] 时，并且不等于 up[3,1,2]，需要向 Monitor 申请 pg_temp 为 want。

5）申请成功 pg_temp 以后，acting 为 [3,1,2],up 为 [1,3,2]，osd1 做为临时的主 OSD，

处理读写请求。当该 PG 恢复处于 clean 状态，pg_temp 取消，acting 和 up 都恢复为 [3,1,2]。

2. find_best_info

函数 find_best_info 用于选取一个拥有权威日志的 OSD。根据 last_epoch_clean 到目前为止，各个 past interval 期间参与该 PG 的所有目前还处于 up 状态的 OSD 上 pg_info_t 信息，来选取一个拥有权威日志的 OSD，选择的优先顺序如下：

1）具有最新的 last_update 的 OSD。

2）如果条件 1 相同，选择日志更长的 OSD。

3）如果 1，2 条件都相同，选择当前的主 OSD。

代码实现具体的过程如下：

1）首先在所有 OSD 中计算 max_last_epoch_started，然后在拥有最大的 last_epoch_started 的 OSD 中计算 min_last_update_acceptable 的值。

2）如果 min_last_update_acceptable 为 eversion_t::max()，返回 infos.end()，选取失败。

3）根据以下条件选择一个 OSD：

a）首先过滤掉 last_update 小于 min_last_update_acceptable，或者 last_epoch_started 小于 max_last_epoch_started_found，或者处于 incomplete 的 OSD。

b）如果 PG 类型是 EC，选择最小的 last_update ；如果 PG 类型是副本，选择最大的 last_update 的 OSD。

c）如果上述条件都相同，选择 long tail 最小的，也就是日志最长的 OSD。

d）如果上述条件都相同，选择当前的主 OSD。

综上的选择过程可知：拥有权威日志的 OSD 特征如下：必须是非 incomplete 的 OSD；必须有最大 last_epoch_strated ；last_update 有可能是最大，但至少是 min_last_update_acceptable，有可能是日志最长的 OSD，有可能是主 OSD。

3. calc_replicated_acting

本函数计算本 PG 相关的下列 OSD 列表：

- want_primary：主 OSD，如果它不是 up_primay，就需要申请 pg_temp。

- backfill：需要进行 Backfill 操作的 OSD。
- acting_backfill：所有进行 acting 和 Backfill 的 OSD 的集合。
- want 和 acting_backfill 的 OSD 相同，前者类型是 pg_shard_t，后者为 int 型。

具体处理过程如下：

1）首先选择 want_primay 列表中的 OSD：

a）如果 up_primary 处于非 incomplete 状态，并且 last_update 大于等于权威日志的 log_tail，说明 up_primary 的日志和权威日志有重叠，可通过日志记录恢复，优先选择 up_primary 为主 OSD。

b）否则选择 auth_log_shard，也就是拥有权威日志的 OSD 为主 OSD。

c）把主 OSD 加入到 want 和 acting_backfill 列表中。

2）函数的输入参数 size 为要选择的副本数，依次从 up、acting、all_info 里选择 size 个副本 OSD：

a）如果该 OSD 上的 PG 处于 incomplete 的状态，或者 cur_info.last_update 小于主 OSD 和 auth_log_shard 的最小值，则该 PG 副本无法通过日志修复，只能通过 Backfill 操作来修复。把该 OSD 分别加入 backfill 和 acting_backfill 集合中。

b）否则就可以根据 PG 日志来恢复，只加入 acting_backfill 集和 want 列表中，不用加入到 Backfill 列表中。

4. 收到缺失的权威日志

如果主 OSD 不是拥有权威日志的 OSD，就需要去拥有权威日志的 OSD 上拉取权威日志：

```
boost::statechart::result PG::RecoveryState::GetLog::react(const MLogRec& logevt)
```

当收到权威日志后，封装成 MLogRec 类型的事件。本函数就用于处理该事件。它首先确认是从 auth_log_shard 端发送的消息，然后抛出 GotLog() 事件：

```
boost::statechart::result PG::RecoveryState::GetLog::react(const GotLog&)
```

本函数捕获 GetLog 事件，处理过程如下：

1）如果 msg 不为空，就调用函数 proc_master_log 合并自己缺失的权威日志，并更新自己 pg_info 相关的信息。从此，做为主 OSD，也是拥有权威日志的 OSD。

2）调用函数 pg->start_flush 添加一个空操作。

3）状态转移到 GetMissing 状态。

经过 GetLog 阶段的处理后，该 PG 的主 OSD 已经获取了权威日志，以及 pg_info 的权威信息。

10.5.7　GetMissing

GetMissing 的处理过程为：首先，拉取各个从 OSD 上的有效日志。其次，用主 OSD 上的权威日志与各个从 OSD 的日志进行对比，从而计算出各从 OSD 上不一致的对象并保存在对应的 pg_missing_t 结构中，做为后续数据修复的依据。

主 OSD 的不一致的对象信息，已经在调用函数 proc_master_log 合并权威日志的过程中计算出来，所以这里只计算从 OSD 上的不一致的对象。

1. 拉取从副本上的日志

在 GetMissing 的构造函数里，通过对比主 OSD 上的权威 pg_info 信息，来获取从 OSD 上的日志信息。

```
PG::RecoveryState::GetMissing::GetMissing(my_context ctx)
```

其具体处理过程为遍历 pg->actingbackfill 的 OSD 列表，然后做如下的处理：

1）不需要获取 PG 日志的情况：

a）如果 pi.is_empty() 为空，没有任何信息，需要 Backfill 过程来修复，不需要获取日志。

b）pi.last_update 小于 pg->pg_log.get_tail()，该 OSD 的 pg_info 记录中，last_update 小于权威日志的尾部记录，该 OSD 的日志和权威日志不重叠，该 OSD 操作已经远远落后于权威 OSD，已经无法根据日志来修复，需要 Backfill 过程来修复。

c）pi.last_backfill 为 hobject_t()，说明在 past interval 期间，该 OSD 标记需要 Backfill 操作，实际并没开始 Backfill 的工作，需要继续 Backfill 过程。

d）pi.last_update 等于 pi.last_complete，说明该 PG 没有丢失的对象，已经完成 Recovery 操作阶段，并且 pi.last_update 等于 pg->info.last_update，说明日志和权威日志的最后更新一致，说明该 PG 数据完整，不需要恢复。

2）获取日志的情况：当 pi.last_update 大于 pg->info.log_tail，该 OSD 的日志记录和权威日志记录重叠，可以通过日志来修复。变量 sine 是从 last_epoch_started 开始的版本值：

a）如果该 PG 的日志记录 pi.log_tail 小于等于版本值 since，那就发送消息 pg_query_t::LOG，从 since 开始获取日志记录。

b）如果该 PG 的日志记录 pi.log_tail 大于版本值 since，就发送消息 pg_query_t::FULLLOG 来获取该 OSD 的全部日志记录。

3）最后检查如果 peer_missing_requested 为空，说明所有获取日志的请求返回并处理完成。如果需要 pg->need_up_thru，抛出 post_event(NeedUpThru())；否则，直接调用 post_event(Activate(pg->get_osdmap()->get_epoch())) 进入 Activate 状态。

下面举例说明获取日志的两种情况：

权威日志	(9, 10)	.....	(10,0) sine	(10,1)	(10,2)	(10,3)	.....	last_update
osd0 日志					log_tail			
osd1 日志	log_tail							

当前 last_epoch_started 的值为 10，sine 是 last_epoch_started 后的首个日志版本值当前需要恢复的有效日志是经过 sine 操作之后的日志，之前的日志已经没有用了。

对应 osd0，其日志 log_tail 小于 since，全部拷贝 osd0 上的日志；对应 osd1，其日志 log_tail 小于 since，只拷贝从 sine 开始的日志记录。

2. 收到从副本上的日志记录处理

当一个 PG 的主 OSD 接收到从 OSD 返回的获取日志 ACK 应答后，就把该消息封装成 MLogRec 事件。状态 GetMissing 接收到该事件后，在下列事件函数里处理该事件：

```
boost::statechart::result PG::RecoveryState::GetMissing::react(const MLogRec&
logevt)
```

具体过程过程如下：

1）调用 proc_replica_log 处理日志。通过日志的对比，获取该 OSD 上处于 missing 状态的对象列表。

2）如果 peer_missing_requested 为空，即所有的获取日志请求返回并处理。如果需要 pg->need_up_thru，抛出 NeedUpThru() 事件。否则，直接调用函数 post_event(Activate(pg->get_osdmap()->get_epoch())) 进入 Activate 状态。

函数 proc_replica_log 处理各个从 OSD 上发过来的日志。它通过比较该 OSD 的日志和本地权威日志，来计算该 OSD 上处于 missing 状态的对象列表。具体处理过程调用 pg_log.proc_replica_log 来处理日志，输出为 omissing，也就是该 OSD 缺失的对象。

10.5.8　Active 操作

由上述可知，如果 GetMissing 处理成功，就跳转到 Activate 状态。到本阶段为止，可以说 Peering 主要工作已经完成，但还需要后续的处理，激活各个副本，如下所示：

```
PG::RecoveryState::Active::Active(my_context ctx)
```

状态 Activate 的构成函数里处理过程如下：

1）在构造函数里初始化了 remote_shards_to_reserve_recovery 和 remote_shards_to_reserve_backfill，需要 Recovery 操作和 Backfill 操作的 OSD。

2）调用函数 pg->start_flush 来完成相关数据的 flush 工作。

3）调用函数 pg->activate 完成最后的激活工作。

1. MissingLoc

类 MissingLoc 用来记录处于 missing 状态对象的位置，也就是缺失对象的正确版本分析在哪些 OSD 上。恢复时就去这些 OSD 上去拉取正确对象的数据：

```
class MissingLoc {
map<hobject_t,pg_missing_t::item, hobject_t::BitwiseComparator> needs_
recovery_map;
   // 缺失的对象 -->  item（现在版本，缺失的版本）
map<hobject_t, set<pg_shard_t>, hobject_t::BitwiseComparator > missing_loc;
  // 缺失的对象 ---> 所在的 OSD 集合
set<pg_shard_t> missing_loc_sources;
```

```
    // 所有缺失对象所在的 OSD 集合
     PG *pg;
     set<pg_shard_t> empty_set;
   public:
     boost::scoped_ptr<IsPGReadablePredicate> is_readable;
boost::scoped_ptr<IsPGRecoverablePredicate> is_recoverable;
}
```

下面介绍一些 MissingLog 处理函数，作用是添加相应的 missing 对象列表。其对应两个函数：add_active_missing 函数和 add_source_info 函数。

add_active_missing 函数用于把一个副本中的所有缺失对象添加到 MissingLoc 的 needs_recovery_map 结构里：

```
void add_active_missing(const pg_missing_t &missing)
```

add_source_info 函数用于计算每个缺失对象是否在本 OSD 上：

```
PG::MissingLoc::add_source_info(pg_shard_t fromosd,
  const pg_info_t &oinfo, const pg_missing_t &omissing,
  bool sort_bitwise,  ThreadPool::TPHandle* handle)
```

具体实现如下：

遍历 needs_recovery_map 里的所有对象，对每个对象做如下处理：

1）如果 oinfo.last_update < need（所需的缺失对象的版本）大于 oinfo.last_update 的值，就跳过。

2）如果该 PG 正常的 last_backfill 指针小于 MAX 值，说明还处于 Backfill 阶段，但是 sort_bitwise 不正确，跳过。

3）如果该对象大于 last_backfill，显然该对象不存在，跳过。

4）如果该对象大于 last_complete，说明该对象或者是上次 Peering 之后缺失的对象，还没有来得及恢复；或者是新创建的对象。检查如果在 missing 记录已存在，就是上次缺失的对象，直接跳过；否则就是新创建的对象，存在该 OSD 中。

5）经过上述检查后，确认该对象在本 OSD 上，在 missing_loc 添加该对象的 location 为本 OSD。

2. Activate 状态

PG::activate 函数是 Peering 过程的最后一步，该函数完成以下功能：

❑ 更新一些 pg_info 的参数信息。

❑ 给 replica 发消息，激活副本 PG。

❑ 计算 MissingLoc，也就是缺失对象分布在哪些 OSD 上，用于后续的恢复。

具体处理过程如下：

1）如果需要客户回答，就把 PG 添加到 replay_queue 队列里。

2）更新 info.last_epoch_started 变量，info.last_epoch_started 指的是本 OSD 在完成目前 Peering 进程后的更新，而 info.history.last_epoch_started 是 PG 的所有的 OSD 都确认完成 Peering 的更新。

3）更新一些相关的字段。

4）注册 C_PG_ActivateCommitted 回调函数，该函数最终完成 activate 的工作。

5）初始化 snap_trimq 快照相关的变量。

6）设置 info.last_complete 指针：

- 如果 missing.num_missing() 等于 0，表明处于 clean 状态。直接更新 info.last_complete 等于 info.last_update，并调用 pg_log.reset_recovery_pointers() 调整 log 的 complete_to 指针。
- 否则，如果有需要恢复的对象，就调用函数 pg_log.activate_not_complete(info)，设置 info.last_complete 为缺失的第一个对象的前一版本。

7）以下都是主 OSD 的操作，给每个从 OSD 发送 MOSDPGLog 类型的消息，激活该 PG 的从 OSD 上的副本。分别对应三种不同处理：

- 如果 pi.last_update 等于 info.last_update，这种情况下，该 OSD 本身就是 clean 的，不需要给该 OSD 发送其他信息。添加到 activator_map 只发送 pg_info 来激活从 OSD。其最终的执行在 PeeringWQ 的线程执行完状态机的事件处理后，在函数 OSD::dispatch_context 里调用 OSD::do_infos 函数实现。
- 需要 Backfill 操作的 OSD，发送 pg_info，以及 osd_min_pg_log_entries 数量的 PG 日志。
- 需要 Recovery 操作的 OSD，发送 pg_info，以及从缺失的日志。

8）设置 MissingLoc，也就是统计缺失的对象，以及缺失的对象所在的 OSD，核心就

是调用 MissingLoc 的 add_source_info 函数，见 MissingLoc 相关的分析。

9）如果需要恢复，把该 PG 加入到 osd->queue_for_recovery(this) 的恢复队列中。

10）如果 PG 的 size 小于 act set 的 size，也就是当前的 OSD 不够，就标记 PG 的状态为 PG_STATE_DEGRADED 和 PG_STATE_UNDERSIZED 状态，最后标记 PG 为 PG_STATE_ACTIVATING 状态。

3. 收到从 OSD 的 MOSDPGLog 的应对

当收到从 OSD 发送的 MOSDPGLog 的 ACK 消息后，触发 MInfoRec 事件，下面这个函数处理该事件：

```
boost::statechart::result PG::RecoveryState::Active::react(const MInfoRec&
  infoevt)
```

处理过程比较简单：检查该请求的源 OSD 在本 PG 的 actingbacfill 列表中，以等待列表中删除该 OSD。最后检查，当收集到所有的从 OSD 发送的 ACK，就调用函数 all_activated_and_committed 触发 AllReplicasActivated 事件。

对应主 OSD 在事务的回调函数 C_PG_ActivateCommitted 里实现，最终调用 _activate_committed 加入 peer_activated 集合里。

4. AllReplicasActivated

这个函数处理 AllReplicasActivated 事件：

```
boost::statechart::result PG::RecoveryState::Active::react(const
  AllReplicasActivated &evt)
```

当所有的 replica 处于 activated 状态时，进行如下处理：

1）取消 PG_STATE_ACTIVATING 和 PG_STATE_CREATING 状态，如果该 PG 上 acting 状态的 OSD 数量大于等于 Pool 的 min_size，设置该 PG 为 PG_STATE_ACTIVE 的状态；否则设置为 PG_STATE_PEERED 状态。

2）ReplicatedPG::check_local 检查本地的 stray 对象是否都被删除。

3）如果有读写请求在等待 Peering 操作，则把该请求添加到处理队列 pg->requeue_ops(pg->waiting_for_peered)。

4）调用函数 ReplicatedPG::on_activate，如果需要 Recovery 操作，触发 DoRecovery 事件，如果需要 Backfill 操作，触发 RequestBackfill 事件；否则触发 AllReplicasRecovered 事件。

5）初始化 Cache Tier 需要的 hit_set 对象。

6）初始化 Cache Tier 需要的 agent 对象。

10.5.9　副本端的状态转移

当创建 PG 后，根据不同的角色，如果是主 OSD，PG 对应的状态机就进入了 Primary 状态。如果不是主 OSD，就进入 Stray 状态。

1. Stray 状态

Stray 状态有两种情况。

情况 1：只接收到 PGINFO 的处理：

```
boost::statechart::result PG::RecoveryState::Stray::react(const MInfoRec& infoevt)
```

从 PG 接收到主 PG 发送的 MInfoRec 事件，也就是接收到主 OSD 发送的 pg_info 信息。其判断如果当前 pg->info.last_update 大于 infoevt.info.last_update，说明当前的日志有 divergent 的日志，调用函数 rewind_divergent_log 清理日志即可。最后抛出 Activate(infoevt.info.last_epoch_started 事件，进入 ReplicaActive 状态。

情况 2：接收到 MOSDPGLog 消息：

```
boost::statechart::result PG::RecoveryState::Stray::react(const MLogRec& logevt)
```

当从 PG 接收到 MLogRec 事件，就对应着接收到主 PG 发送的 MOSDPGLog 消息，其通知从 PG 处于 activate 状态，具体处理过程如下：

1）如果 msg->info.last_backfill 为 hobject_t()，需要 Backfill 操作的 OSD。

2）否则就是需要 Recovery 操作的 OSD，调用 merge_log 把主 OSD 发送过来的日志合并。

抛出 Activate(logevt.msg->info.last_epoch_started) 事件，使副本转移到 ReplicaActive 状态。

2. ReplicateActive 状态

ReplicateActive 状态如下：

```
boost::statechart::result PG::RecoveryState::ReplicaActive::react(
  const Activate& actevt)
```

当处于 ReplicateActive 状态，接收到 Activate 事件，就调用函数 pg->activate，在函数 _activate_committed 给主 PG 发送应答信息，告诉自己处于 activate 状态，设置 PG 为 activate 状态。

10.5.10 状态机异常处理

在上面的流程介绍中，只介绍了正常状态机的转换流程。Ceph 之所以用状态机来实现 PG 的状态转换，就是可以实现任何异常情况下的处理。下面介绍当 OSD 失效时导致相关的 PG 重新进行 Peering 的机制。

当一个 OSD 失效，Monitor 会通过 heartbeat 检测到，导致 osd map 发生了变化，Monitor 会把最新的 osd map 推送给 OSD，导致 OSD 上的受影响 PG 重新进行 Peering 操作。

具体的流程如下：

1）在函数 OSD::handle_osd_map 处理 osd map 的变化，该函数调用 consume_map，对每一个 PG 调用 pg->queue_null，把 PG 加入到 peering_wq 中。

2）peering_wq 的处理函数 process_peering_events 调用 OSD::advance_pg 函数，在该函数里调用 PG::handle_advance_map 给 PG 的状态机 RecoveryMachine 发送 AdvMap 事件：

```
boost::statechart::result PG::RecoveryState::Started::react(const AdvMap& advmap)
```

当处于 Started 状态，接收到 AdvMap 事件，调用函数 pg->should_restart_peering 检查，如果是 new_interval，就跳转到 Reset 状态，重新开始一次 Peering 过程。

10.6 本章小结

本章介绍了 Ceph 的 Peering 过程，其核心过程就是通过各个 OSD 上保存的 PG 日志选择出一个权威日志的 OSD。以该 OSD 上的日志为基础，对比其他 OSD 上的日志记录，计算出各个 OSD 上缺失的对象信息。这样，PG 就使各个 OSD 的数据达成了一致。

第 11 章 Chapter 11

Ceph 数据修复

当 PG 完成了 Peering 过程后，处于 Active 状态的 PG 就已经可以对外提供服务了。如果该 PG 的各个副本上有不一致的对象，就需要进行修复。Ceph 的修复过程有两种：Recovery 和 Backfill。

Recovery 是仅依据 PG 日志中的缺失记录来修复不一致的对象。Backfill 是 PG 通过重新扫描所有的对象，对比发现缺失的对象，通过整体拷贝来修复。当一个 OSD 失效时间过长导致无法根据 PG 日志来修复，或者新加入的 OSD 导致数据迁移时，就会启动 Backfill 过程。

从第 10 章可知，PG 完成 Peering 过程后，就处于 activate 状态，如果需要 Recovery，就产生 DoRecovery 事件，触发修复操作。如果需要 Backfill，就会产生 RequestBackfill 事件来触发 Backfill 操作。在 PG 的数据修复过程中，如果既有需要 Recovery 过程的 OSD，又有需要 Backfill 过程的 OSD，那么处理过程需要先进行 Recovery 过程的修复，再完成 Backfill 过程的修复。

本章介绍 Ceph 的数据修复的实现过程。首先介绍数据修复的资源预约的知识，然后通过介绍修复的状态转换图，大概了解整个数据修复的过程。最后分别详细介绍 Recovery 过程和 Backfill 过程的具体实现。

11.1 资源预约

在数据修复的过程中，为了控制一个 OSD 上正在修复的 PG 最大数目，需要资源预约，在主 OSD 上和从 OSD 上都需要预约。如果没有预约成功，需要阻塞等待。一个 OSD 能同时修复的最大 PG 数在配置选项 osd_max_backfills 中设置，默认值为 1。

类 AsyncReserver 用来管理资源预约，其模板参数 <T> 为要预约的资源类型。该类实现了异步的资源预约。当成功完成资源预约后，就调用注册的回调函数通知调用方预约成功：

```
class AsyncReserver {
  unsigned max_allowed;    // 定义允许的最大资源数量，在这里指允许修复的 PG 的数量
  unsigned min_priority;   // 最小的优先级
  Finisher *f;             // 当预约成功够，用来执行的回调函数

  map<unsigned, list<pair<T, Context*> > > queues;
  // 优先级到待预约资源链表的映射，pair<T, Context*> 定义预约的资源和注册的回调函数
 map<T, pair<unsigned, typename list<pair<T, Context*> >::iterator > > queue_
pointers;                  // 资源在 queue 链表中的位置指针
  set<T> in_progress;      // 预约成功，正在使用的资源
}
```

1. 资源预约

函数 request_reservation 用于预约资源：

```
void request_reservation(
  T item,                        // 输入参数：申请的资源
  Context *on_reserved,          // 输入参数：申请成功后的回调函数
```

具体处理过程如下：

1）把要请求的资源根据优先级添加到 queue 队列中，并在 queue_pointers 中添加其对应的位置指针：

```
queues[prio].push_back(make_pair(item, on_reserved));
queue_pointers.insert(make_pair(item,make_pair(prio,--(queues[prio]).
  end())));
```

2）调用函数 do_queues 用来检查 queue 中的所有资源预约申请：从优先级高的请求开始检查，如果还有配额并且其请求的优先级至少不小于最小优先级，就把资源授权给它。

3）在 queue 队列中删除该资源预约请求，并在 queue_pointers 删除该资源的位置信息。把该资源添加到 in_progress 队列中，并把请求相应的回调函数添加到 Finisher 类中，使其执行该回调函数。最后通知预约成功。

2. 取消预约

函数 cancel_reservation 用于释放拥有的不再使用的资源：

```
void cancel_reservation(
  T item                    // 输入参数：删除申请的资源
  )
```

具体处理过程如下：

1）如果该资源还在 queue 队列中，就删除（这属于异常情况的处理）；否则在 in_progress 队列中删除该资源。

2）调用 do_queues 函数把该资源重新授权给其他等待的请求。

11.2　数据修复状态转换图

如图 11-1 所示的是修复过程状态转换图。当 PG 进入 Active 状态后，就进入默认的子状态 Activating。

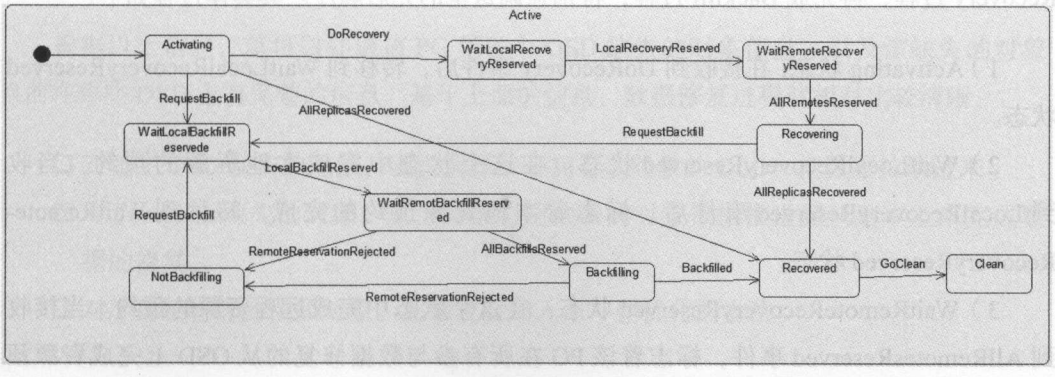


图 11-1　修复过程状态转换图

数据修复的状态转换过程如下所示。

情况 1：当进入 Activating 状态后，如果此时所有的副本都完整，不需要修复，其状态转移过程如下：

1）Activating 状态接收到 AllReplicatedRecovered 事件，直接转换到 Recovered 状态。

2）Recovered 状态接收到 GoClean 事件，整个 PG 转入 Clean 状态。

情况 2：当进入 Activating 状态后，没有 Recovery 过程，只需要 Backfill 过程的情况：

1）Activating 状态直接接收到 RequestBackfill 事件，进入 WaitLocalBackfillReservede 状态。

2）当 WaitLocalBackfillReservede 状态接收到 LocalBackfillReserved 事件后，意味着本地资源预约成功，转入 WaitRemoteBackfillReserved 状态。

3）所有副本资源预约成功后，主 PG 就会接收到 AllBackfillsReserved 事件，进入 Backfilling 状态，开始实际数据 Backfill 操作过程。

4）Backfilling 状态接收 Backfilled 事件，标志 Backfill 过程完成，进入 Recovered 状态。

5）异常处理：当在状态 WaitRemotBackfillReserved 和 Backfilling 接收到 Remote ReservationRejected 事件时，表明资源预约失败，进入 NotBackfilling 状态，再次等待 RequestBackfilling 事件来重新发起 Backfill 过程。

情况 3：当 PG 既需要 Recovery 过程，也可能需要 Backfill 过程时，PG 先完成 Recovery 过程，再完成 Backfill 过程，特别强调这里的先后顺序。其具体过程如下：

1）Activating 状态：在接收到 DoRecovery 事件后，转移到 WaitLocalRecoveryReserved 状态。

2）WaitLocalRecoveryReserved 状态：在这个状态中完成本地资源的预约。当收到 LocalRecoveryReserved 事件后，标志着本地资源预约的完成，转移到 WaitRemote-RecoveryReserved 状态。

3）WaitRemoteRecoveryReserved 状态：在这个状态中完成远程资源的预约。当接收到 AllRemotesReserved 事件，标志着该 PG 在所有参与数据修复的从 OSD 上完成资源预约，进入 Recoverying 状态。

4）Recoverying 状态：在这个状态中完成实际的数据修复工作。完成后把 PG 设置为

PG_STATE_RECOVERING 状态，并把 PG 添加到 recovery_wq 工作队列中，开始启动数据修复：

```
pg->state_clear(PG_STATE_RECOVERY_WAIT);
pg->state_set(PG_STATE_RECOVERING);
pg->osd->queue_for_recovery(pg);
```

5）在 Recoverying 状态完成 Recovery 工作后，如果需要 Backfill 工作，就接收 RequestBackfill 事件，转入 Backfill 流程。

6）如果没有 Backfill 工作流程，直接接收 AllReplicasRecovered 事件，转入 Recovered 状态。

7）Recovered 状态：到达本状态，意味着已经完成数据修复工作。当收到事件 GoClean 后，PG 就进入 clean 状态。

11.3　Recovery 过程

数据修复的依据是在 Peering 过程中产生的如下信息：

- 主副本上的缺失对象的信息保存在 pg_log 类的 pg_missing_t 结构中。
- 各从副本上的缺失对象信息保存在 OSD 对应的 peer_missing 中的 pg_missing_t 结构中。
- 缺失对象的位置信息保存在类 MissingLoc 中。

根据以上信息，就可以知道该 PG 里各个 OSD 缺失的对象信息，以及该缺失的对象目前在哪些 OSD 上有完整的信息。基于上面的信息，数据修复过程就相对比较清晰：

- 对于主 OSD 缺失的对象，随机选择一个拥有该对象的 OSD，把数据拉取过来。
- 对于 replica 缺失的对象，从主副本上把缺失的对象数据推送到从副本上来完成数据的修复。
- 对于比较特殊的快照对象，在修复时加入了一些优化的方法。

11.3.1　触发修复

Recovery 过程由 PG 的主 OSD 来触发并控制整个修复的过程。在修复的过程中，先

修复主 OSD 上缺失（或者不一致）的对象，然后修复从 OSD 上缺失的对象。由数据修复状态转换过程可知，当 PG 处于 Activate/Recoverying 状态后，该 PG 被加入到 OSD 的 RecoveryWQ 工作队列中。在 recovery_wq 里，其工作队列的线程池的处理函数调用 do_recovery 函数来执行实际的数据修复操作：

```
void OSD::do_recovery(PG *pg, ThreadPool::TPHandle &handle)
```

函数 do_recovery 由 RecoveryWQ 工作队列的线程池的线程执行。其输入的参数为要修复的 PG，具体处理流程如下：

1）配置选项 osd_recovery_sleep 设置了线程做一次修复后的休眠时间。如果设置了该值，每次线程开始先休眠相应的时间长度。该参数默认值为 0，不需要休眠。

2）加 recovery_wq.lock() 锁，用来保护 recovery_wq 队列以及变量 recovery_ops_active。计算可修复对象的 max 值，其值为允许修复的最大对象数 osd_recovery_max_active 减去正在修复的对象数 recovery_ops_active，然后调用函数 recovery_wq.unlock() 解锁。

3）如果 max 小于等于 0，即没有修复对象的配额，就把 PG 重新加入工作队列 recovery_wq 中并返回；否则如果 max 大于 0，调用 pg->lock_suspend_timeout(handle) 重新设置线程超时时间。检查 PG 的状态，如果该 PG 处于正在被删除状态，或者既不处于 peered 状态，也不是主 OSD，则直接退出。

4）调用函数 pg->start_recovery_ops 修复，返回值 more 为还需要修复的对象数目。输出参数 started 为已经开始修复的对象数。

5）如果 more 为 0，也就是没有修复的对象了。但是 pg->have_unfound() 不为 0，还有 unfound 对象（即缺失的对象，目前不知道在哪个 OSD 上能找到完整的对象），调用函数 discover_all_missing 在 might_have_unfound 队列中的 OSD 上继续查找该对象，查找的方法就是给相关的 OSD 发送获取该 OSD 的 pg_log 的消息。

6）如果 rctx.query_map->empty() 为空，也就是没有找到其他 OSD 去获取 pg_log 来查找 unfound 对象，就结束该 PG 的 recover 操作，调用函数从 recovery_wq._dequeue(pg) 删除 PG。

7）函数 dispatch_context 做收尾工作，在这里发送 query_map 的请求，把 ctx.transaction 的事务提交到本地对象存储中。

由上过程分析可知，do_recovery 函数的核心功能是计算要修复对象的 max 值，然后

调用函数 start_recovery_ops 来启动修复。

11.3.2　ReplicatedPG

类 ReplicatedPG 用于处理 Replicate 类型 PG 的相关修复操作。下面分析它用于修复的 start_recovery_ops 函数及其相关函数的具体实现。

1. start_recovery_ops

函数 start_recovery_ops 调用 recovery_primary 和 recovery_replicas 来修复该 PG 上对象的主副本和从副本。修复完成后，如果仍需要 Backfill 过程，则抛出相应事件触发 PG 状态机，开始 Backfill 的修复进程。

```
bool ReplicatedPG::start_recovery_ops(int max, RecoveryCtx *prctx,
  ThreadPool::TPHandle &handle,int *ops_started)
```

该函数具体处理过程如下：

1）首先检查 OSD，确保该 OSD 是 PG 的主 OSD。如果 PG 已经处于 PG_STATE_RECOVERING 或者 PG_STATE_BACKFILL 的状态则退出。

2）从 pg_log 获取 missing 对象，它保存了主 OSD 缺失的对象。参数 num_missing 为主 OSD 缺失的对象数目；num_unfound 为该 PG 上缺失的对象却没有找到该对象其他正确副本所在的 OSD；如果 num_missing 为 0，说明主 OSD 不缺失对象，直接设置 info.last_complete 为最新版本 info.last_update 的值。

3）如 num_missing 等于 num_unfound，说明主 OSD 所缺失对象都为 unfound 类型的对象，先调用函数 recover_replicas 启动修复 replica 上的对象。

4）如果 started 为 0，也就是已经启动修复的对象数量为 0，调用函数 recover_primary 修复主 OSD 上的对象。

5）如果 started 仍然为 0，且 num_unfound 有变化，再次启动 recover_replicas 修复副本。

6）如果 started 不为零，设置 work_in_progress 的值为 true。

7）如果 recovering 队列为空，也就是没有正在进行 Recovery 操作的对象，状态为 PG_STATE_BACKFILL，并且 backfill_targets 不为空，started 小于 max，missing.num_missing()

为 0，的情况下：

a）如果标志 get_osdmap()->test_flag(CEPH_OSDMAP_NOBACKFILL) 设置了，就推迟 Backfill 过程。

b）如果标志 CEPH_OSDMAP_NOREBALANCE 设置了，且是 degrade 的状态，推迟 Backfill 过程。

c）如果 backfill_reserved 没有设置，就抛出 RequestBackfill 事件给状态机，启动 Backfill 过程。

d）否则，调用函数 recover_backfill 开始 Backfill 过程。

8）最后 PG 如果处于 PG_STATE_RECOVERING 状态，并且对象修复成功，就检查：如果需要 Backfill 过程，就向 PG 的状态机发送 RequestBackfill 事件；如果不需要 Backfill 过程，就抛出 AllReplicasRecovered 事件。

9）否则，PG 的状态就是 PG_STATE_BACKFILL 状态，清除该状态，抛出 Backfilled 事件。

2. recover_primary

函数 recover_primary 用来修复一个 PG 的主 OSD 上缺失的对象：

```
int ReplicatedPG::recover_primary(int max, ThreadPool::TPHandle &handle)
```

其处理过程如下：

1）调用 pgbackend->open_recovery_op 返回一个 PG 类型相关的 PGBackend::RecoveryHandle。对于 ReplicatedPG 对应的 RPGHandle，内部有两个 map，保存了 Push 和 Pull 操作的封装 PushhOp 和 PullOP：

```
struct RPGHandle : public PGBackend::RecoveryHandle {
  map<pg_shard_t, vector<PushOp> > pushes;
  map<pg_shard_t, vector<PullOp> > pulls;
};
```

2）last_requested 为上次修复的指针，通过调用 low_bound 函数来获取还没有修复的对象。

3）遍历每一个未被修复的对象：latest 为日志记录中保存的该缺失对象的最后的一条日志，soid 为缺失的对象。如果 latest 不为空：

a）如果该日志记录是 pg_log_entry_t::CLONE 类型，这里不做任何的特殊处理，直到成功获取 snapshot 相关的信息 SnapSet 后再处理。

b）如果该日志记录类型为 pg_log_entry_t::LOST_REVERT 类型：该 revert 操作为数据不一致时，管理员通过命令行强行回退到指定版本，reverting_to 记录了回退的版本号：

- 如果 item.have 等于 latest->reverting_to 版本，也就是通过日志记录显示当前已经拥有回退的版本，那么就获取对象的 ObjectContext，如果检查对象当前的版本 obc->obs.oi.version 等于 latest->version，说明该回退操作完成。
- 如果 item.have 等于 latest->reverting_to，但是对象当前的版本 obc->obs.oi.version 不等于 latest->version，说明没有执行回退操作，直接修改对象的版本号为 latest->version 即可。
- 否则，需要拉取该 reverting_to 版本的对象，这里不做特殊的处理，只是检查所有 OSD 是否拥有该版本的对象，如果有就加入到 missing_loc 记录该版本的位置信息，由后续修复继续来完成。

c）如果该对象在 recovering 过程中，表明正在修复，或者其 head 对象正在修复，跳过，并计数增加 skipped；否则调用函数 recover_missing 来修复。

4）调用函数 pgbackend->run_recovery_op，把 PullOp 或者 PushOp 封装的消息发送出去。

下面举例说明，当最后的日志记录类型为 LOST_REVERT 时的修复过程。

例 11-1　日志修复过程。

PG 日志的记录如下：每个单元代表一条日志记录，分别为对象的名字和版本以及操作，版本的格式为（epoch,version）。灰色的部分代表本 OSD 上缺失的日志记录，该日志记录是从权威日志记录中拷贝过来的，所以当前该日志记录是连续完整的。

obj2(1,3) modify	obj1(1,4) modify	obj2(1,5) modify	obj1(1,6) modify	obj1(1,7) modify	obj1(1,8) modify

情况 1：正常情况的修复。

缺失的对象列表为 [obj1，obj2]。当前修复对象为 obj1。由日志记录可知：对象 obj1 被修改过三次，分别为版本 6,7,8。当前拥有的 obj1 对象的版本 have 值为 4，修复时只修复到最后修改的版本 8 即可。

情况 2：最后一个操作为 LOST_REVERT 类型的操作。

obj2(1,3) modify	obj1(1,4) modify	obj2(1,5) modify	obj1(1,6) modify	obj1(1,7) modify	obj1(1,8) lost_revert_ version = 8 prior_version=7 reverting_to=4

对于要修复的对象 obj1，最后一次操作为 LOST_REVERT 类型的操作，该操作当前版本 version 为 8，修改前的版本 prior_version 为 7，回退版本 reverting_to 为 4。

在这种情况下，日志显示当前已经有版本 4，检查对象 obj1 的实际版本，也就是 object_info 里保存的版本号：

1）如果该值是 8，说明最后一次 revert 操作成功，不需要做任何修复动作。

2）如果该值是 4，说明 LOST_REVERT 操作就没有执行。当然数据内容已经是版本 4 了，只需要修改 object_info 的版本为 8 即可。

如果回退的版本 reverting_to 不是版本 4，而是版本 6，那么最终还是需要把 obj1 的数据修复到版本 6 的数据。Ceph 在这里的处理，仅仅是检查其他 OSD 缺失的对象中是否有版本 6，如果有，就加入到 missing_loc 中，记录拥有该版本的 OSD 位置，待后续继续修复。

3. recover_missing

函数 recovery_missing 处理 snap 对象的修复。在修复 snap 对象时，必须首先修复 head 对象或者 snapdir 对象，获取 SnapSet 信息，然后才能修复快照对象自己。

```
int ReplicatedPG::recover_missing(const hobject_t &soid, eversion_t v, int
  priority, PGBackend::RecoveryHandle *h)
```

具体实现如下：

1）检查如果对象 soid 是 unfound，直接返回 PULL_NONE 值。暂时无法修复处于 unfound 的对象。

2）如果修复的是 snap 对象：

a）查看如果对应的 head 对象处于 missing，递归调用函数 recover_missing 先修复 head 对象。

b）查看如果 snapdir 对象处于 missing，就递归调用函数 recover_missing 先修复 snapdir 对象。

3）从 head 对象或者 snapdir 对象中获取 head_obc 信息。

4）调用函数 pgbackend->recover_object 把要修复的操作信息封装到 PullOp 或者 PushOp 对象中，并添加到 RecoveryHandle 结构中。

11.3.3　pgbackend

pgbackend 封装了不同类型的 Pool 的实现。ReplicatedBackend 实现了 replicate 类型的 PG 相关的底层功能，ECbackend 实现了 Erasure code 类型的 PG 相关的底层功能。

由 11.3.2 节的分析可知，需要调用 pgbackend 的 recovery_object 函数来实现修复对象的信息封装。这里只介绍基于副本的。

函数 recovery_object 实现 pull 操作，调用 prepare_pull 操作把请求封装成 PullOp 结构。如果是 push 操作，就调用 start_pushes 把请求封装成 PushOp 的操作。

1. pull 操作

Prepare_pull 函数把要拉取的 object 相关的操作信息打包成 PullOp 类信息，如下所示：

```
void ReplicatedBackend::prepare_pull(
  eversion_t v,                    // 要拉取对象的版本信息
  const hobject_t& soid,           // 要拉取的对象
  ObjectContextRef headctx,        // 拉取对象的 ObjectContext 信息
  RPGHandle *h)                    // 封装后保存的 RecoveryHandle
```

难点在于 snap 对象的修复处理过程。下面先介绍 PullOp 数据结构。

PullOp 数据结构如下：

```
struct PullOp {
  hobject_t soid;                                    // 需要拉取的对象

  ObjectRecoveryInfo recovery_info;                  // 对象修复的信息
  ObjectRecoveryProgress recovery_progress;          // 对象修复进度信息
}

struct ObjectRecoveryInfo {
  hobject_t soid;                                    // 修复的对象
  eversion_t version;                                // 修复对象的版本
  uint64_t size;                                     // 修复对象的大小
  object_info_t oi;                                  // 修复对象的 object_info 信息
  SnapSet ss;                                        // 修复对象的快照信息

  interval_set<uint64_t> copy_subset;
// 对象需要拷贝的集合，在修复快照对象时，需要从别的 OSD 拷贝到本地的对象的区段集合
  map<hobject_t, interval_set<uint64_t> > clone_subset;
  //clone 对象修复时，需要从本地对象拷贝来修复的区间
}

struct ObjectRecoveryProgress {
  bool first;                                       // 是否是首次修复操作
  uint64_t data_recovered_to;                       // 数据已经修复的位置指针
  bool data_complete;                               // 数据是否修复完成
  string omap_recovered_to;                         //omap 已经修复的位置指针
  bool omap_complete;                               //omap 是否修复完成
}
```

函数 prepare_pull 具体处理过程如下：

1）通过调用函数 get_parent() 来获取 PG 对象的指针。pgbackend 的 parent 就是相应的 PG 对象。通过 PG 获取 missing、peer_missing、missing_loc 等信息。

2）从 soid 对象对应的 missing_loc 的 map 中获取该 soid 对象所在的 OSD 集合。把该集合保存在 shuffle 这个向量中。调用 random_shuffle 操作对 OSD 列表随机排序，然后选择向量中首个 OSD 作为缺失对象来拉取源 OSD 的值。从这一步可知，当修复主 OSD 上的对象，而多个从 OSD 上有该对象时，随机选择其中一个源 OSD 来拉取。

3）当选择了一个源 shard 之后，查看该 shard 对应的 peer_missing 来确保该 OSD 上不缺失该对象，即确实拥有该版本的对象。

4）确定拉取对象的数据范围：

a）如果是 head 对象，直接拷贝对象的全部，在 copy_subset 加入区间（0，-1），表示全部拷贝，最后设置 size 为 -1：

```
recovery_info.copy_subset.insert(0, (uint64_t)-1);
recovery_info.size = ((uint64_t)-1);
```

b）如果该对象是 snap 对象，确保 head 对象或者 snapdir 对象二者必须存在一个。如果 headctx 不为空，就可以获取 SnapSetContext 对象，它保存了 snapshot 相关的信息。调用函数 calc_clone_subsets 来计算需要拷贝的数据范围。

5）设置 PullOp 的相关字段，并添加到 RPGHandle 中。

函数 calc_clone_subsets 用于修复快照对象。在介绍它之前，这里需要介绍 SnapSet 的数据结构和 clone 对象的 overlap 概念。

在 SnapSet 结构中，字段 clone_overlap 保存了 clone 对象和上一次 clone 对象的重叠的部分：

```
struct SnapSet {
  snapid_t seq;
  bool head_exists;
  vector<snapid_t> snaps;     // 序号降序排列
  vector<snapid_t> clones;    // 序号升序排列
  map<snapid_t, interval_set<uint64_t> > clone_overlap;
                              // 写操作导致的和最新的克隆对象重叠的部分
  map<snapid_t, uint64_t> clone_size;
};
```

下面通过一个示例来说明 clone_overlap 数据结构的概念。

例 11-2　clone_overlap 数据结构如图 11-2 所示：

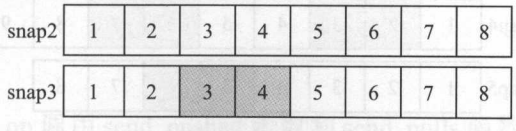


图 11-2　clone_overlap 示意图

snap3 从 snap2 对象 clone 出来，并修改了区间 3 和 4，其在对象中范围的 offset 和 length 为（4,8）和（8,12）。那么在 SnapSet 的 clone_overlap 中就记录：

```
clone_overlap[3] = { 4,8),(8,12)}
```

函数 calc_clone_subsets 用于修复快照对象时，计算应该拷贝的数据区间。在修复快照对象时，并不是完全拷贝快照对象，这里用于优化的关键在于：快照对象之间是有数据重叠，数据重叠的部分可以通过已存在的本地快照对象的数据拷贝来修复；对于不能通过本地快照对象拷贝修复的部分，才需要从其他副本上拉取对应的数据。

函数 calc_clone_subsets 具体实现如下：

1）首先获取该快照对象的 size，把（0,size）加入到 data_subset 中：

```
data_subset.insert(0, size);
```

2）向前查找（oldest snap）和当前快照相交的区间，直到找到一个不缺失的快照对象，添加到 clone_subsets 中。这里找的不重叠区间，是从不缺失快照对象到当前修复的快照对象之间从没有修改过的区间，所以修复时，直接从已存在的快照对象拷贝所需区间数据即可。

3）同理，向后查找（newset snap）和当前快照对象相重叠的对象，直到找到一个不缺失的对象，添加到 clone_subsets 中。

4）去除掉所有重叠的区间，就是需要拉取的数据区间：

```
data_subset.subtract(cloning);
```

对于上述的算法，下面举例来说明。

例 11-3 快照对象修复示例如图 11-3 所示：

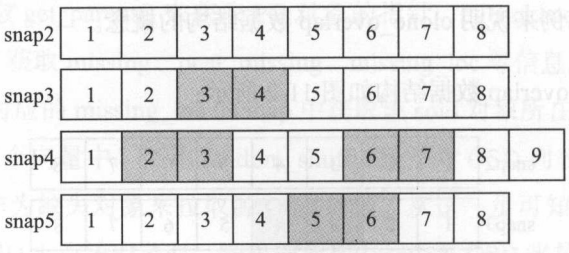


图 11-3 快照对象修复计算示例图

要修复的对象为 snap4，不同长度代表各个 clone 对象的 size 是不同的，其中灰色的区间代表 clone 后修改的区间。snap2、snap3 和 snap5 都是已经存在的非缺失对象。

算法处理流程如下：

1）向前查找和 snap4 重叠的区间，直到遇到非缺失对象 snap2 为止。从 snap4 到 snap2 一直重叠的区间为 1,5,8 三个区间。因此，修复对象 snap4 时，修复 1,5,8 区间的数据，可以直接从已经存在的本地非缺失对象 snap2 拷贝即可。

2）同理，向后查找和 snap4 重叠的区间，直到遇到非缺失对象 snap5 为止。snap5 和 snap4 重叠的区间为 1,2,3,4,7,8 六个区间。因此，修复对象 snap4 时，直接从本地对象 snap4 中拷贝区间 1,2,3,4,7,8 即可。

3）去除上述本地就可修复的区间，对象 snap4 只有区间 6 需要从其他 OSD 上拷贝数据来修复。

2. push 操作

函数 start_pushes 获取 actingbackfill 的 OSD 列表，通过 peer_missing 查找缺失该对象的 OSD，调用 prep_push_to_replica 打包 PushOp 请求。

函数 prep_push_to_replica 实现过程如下：

1）如果需要 push 的对象是 snap 对象：检查如果 head 对象缺失，调用 prep_push 推送 head 对象；如果是 headdir 对象缺失，则调用 prep_push 推送 headdir 对象。

2）如果是 snap 对象，调用函数 calc_clone_subsets 来计算需要推送的快照对象的数据区间。

3）如果是 head 对象，调用 calc_head_subsets 来计算需要推送的 head 对象的区间，其原理和计算快照对象类似，这里就不详细说明了。最后调用 prep_push 封装 PushInfo 信息，在函数 build_push_op 里读取要 push 的实际数据。

3. 处理修复操作

函数 run_recover_op 调用 send_pushed 函数和 send_pulls 函数把请求发送给相关的 OSD，这个流程比较简单。

当主 OSD 把对象推送给缺失该对象的从 OSD 后，从 OSD 需要调用函数 handle_push 来实现数据写入工作，从而来完成该对象的修复。同样，当主 OSD 给从 OSD 发起拉取对

象的请求来修复自己缺失的对象时，需要调用函数 handle_pulls 来处理该请求的应对。

在函数 ReplicatedBackend::handle_push 里处理 handle_push 的请求，主要调用 submit_push_data 函数来写入数据。

handle_pulls 函数收到一个 PullOp 操作，返回 PushOp 操作，处理流程如下：

1）首先调用 store->stat 函数，验证该对象是否存在，如果不存在，则调用函数 prep_push_op_blank，直接返回空值。

2）如果该对象存在，获取 ObjectRecoveryInfo 和 ObjectRecoveryProgress 结构。如果 progress.first 为 true 并且 recovery_info.size 为 –1，说明是全拷贝修复：将 recovery_info.size 设置为实际对象的 size，清空 recovery_info.copy_subset，并把 (0,size) 区间添加到 recovery_info.copy_subset.insert(0, st.st_size) 的拷贝区间。

3）调用函数 build_push_op，构建 PullOp 结构。如果出错，调用 prep_push_op_blank，直接返回空值。

函数 build_push_on 完成构建 push 的请求。具体处理如下：

1）如果 progress.first 为 true，就需要获取对象的元数据信息。通过 store->omap_get_header 获取 omap 的 header 信息，通过 store->getattrs 获取对象的扩展属性信息，并验证 oi.version 是否为 recovery_info.version ；否则返回 -EINVAL 值。如果成功，new_progress.first 设置为 false。

2）上一步只是获取了 omap 的 header 信息，并没有获取 omap 信息。这一步首先判断 progress.omap_complete 是否完成，（初始化设置为 false）如果没有完成，就迭代获取 omap 的（key,value）信息，并检查一次获取信息的大小不能超过 cct->_conf->osd_recovery_max_chunk 设置的值（默认为 8MB）。特别需要注意的是，当该配置参数的值小于一个对象的 size 时，一个对象的修复需要多次数据的 push 操作。为了保证数据的完整一致性，先把数据拷贝到 PG 的 temp 存储空间。当拷贝完成之后，再移动到该 PG 的实际空间中。

3）开始拷贝数据：检查 recovery_info.copy_subset，也就是拷贝的区间。

4）调用函数 store->fiemap 来确定有效数据的区间 out_op->data_included 的值，通过 store->read 读取相应的数据到 data 里。

5）设置 PullOp 的相关的字段，并返回。

11.4　Backfill 过程

当 PG 完成了 Recovery 过程之后，如果 backfill_targets 不为空，表明有需要 Backfill 过程的 OSD，就需要启动 Backfill 的任务，来完成 PG 的全部修复。下面介绍 Backfill 过程相关的数据结构和具体处理过程。

11.4.1　相关数据结构

数据结构 BackfillInterval 用来记录每个 peer 上的 Backfill 过程。其字段说明如下：

- version：记录扫描对象列表时，当前 PG 对象更新的最新版本，一般为 last_update，由于此时 PG 处于 active 状态，可能正在进行写操作。其用来检查从上次扫描到现在是否有对象写操作。如果有，完成写操作的对象在已扫描的对象列表中，进行 Backfill 操作时，该对象就需要更新为最新版本。
- objects：扫描到的准备进行 Backfill 操作的对象列表。
- begin：当前处理的对象。
- end：本次扫描对象的结束，用于作为下次扫描对象的开始：

```
struct BackfillInterval {
  // 一个 peer 的 backfill_interval 信息
  eversion_t version; // 扫描时的最新对象版本
  map<hobject_t,eversion_t,hobject_t::Comparator> objects;
  bool sort_bitwise;
  hobject_t begin;     // 当前处理的对象
  hobject_t end;       // 本次扫描对象的结束
}
```

11.4.2　Backfill 的具体实现

函数 recovery_backfill 作为 Backfill 过程的核心函数，控制整个 Backfill 修复进程。其工作流程如下。

1）初始设置。

在函数 on_activate 里设置了 PG 的属性值 new_backfill 为 true，设置了 last_backfill_started 为 earliest_backfill() 的值。该函数计算需要 backfill 的 OSD 中，peer_info 信息里保存的 last_backfill 的最小值。

peer_backfill_info 的 map 中保存各个需要 Backfill 的 OSD 所对应 backfillInterval 对象信息。首先初始化 begin 和 end 都为 peer_info.last_backfill，由 PG 的 Peering 过程可知，在函数 activate 里，如果需要 Backfill 的 OSD，设置该 OSD 的 peer_info 的 last_backfill 为 hobject_t()，也就是 MIN 对象。

backfills_in_flight 保存了正在进行 Backfill 操作的对象，pending_backfill_updates 保存了需要删除的对象。

2）设置 backfill_info.begin 为 last_backfill_started，调用函数 update_range 来更新需要进行 Backfill 操作的对象列表。

3）根据各个 peer_info 的 last_backfill 对相应的 backfillInterval 信息进行 trim 操作。根据 last_backfill_started 来更新 backfill_info 里相关字段。

4）如果 backfill_info.begin 小于等于 earliest_peer_backfill()，说明需要继续扫描更多的对象，backfill_info 重新设置，这里特别注意的是，backfill_info 的 version 字段也重新设置为（0,0），这会导致在随后调用的 update_scan 函数再调用 scan_range 函数来扫描对象。

5）进行比较，如果 pbi.begin 小于 backfill_info.begin，需要向各个 OSD 发送 MOSDPGScan::OP_SCAN_GET_DIGEST 消息来获取该 OSD 目前拥有的对象列表。

6）当获取所有 OSD 的对象列表后，就对比当前主 OSD 的对象列表来进行修复。

7）check 对象指针，就是当前 OSD 中最小的需要进行 Backfill 操作的对象：

a）检查 check 对象，如果小于 Backfill_info.begin，就在各个需要 Backfill 操作的 OSD 上删除该对象，加入到 to_remove 队列中。

b）如果 check 对象大于或者等于 backfill_info.begin，检查拥有 check 对象的 OSD，如果版本不一致，加入 need_ver_targ 中。如果版本相同，就加入 keep_ver_targs 中。

c）那些 begin 对象不是 check 对象的 OSD，如果 pinfo.last_backfil 小于 backfill_info.begin，那么，该对象缺失，加入 missing_targs 列表中。

d）如果 pinfo.last_backfil 大于 backfill_info.begin，说明该 OSD 修复的进度已经超越当前的主 OSD 指示的修复进度，加入 skip_targs 中。

8）对于 keep_ver_targs 列表中的 OSD，不做任何操作。对于 need_ver_targs 和 missing_targs 中的 OSD，该对象需要加入到 to_push 中去修复。

9）调用函数 send_remove_op 给 OSD 发送删除的消息来删除 to_remove 中的对象。

10）调用函数 prep_backfill_object_push 把操作打包成 PushOp，调用函数 pgbackend->run_recovery_op 把请求发送出去。其流程和 Recovery 流程类似。

11）最后用 new_last_backfill 更新各个 OSD 的 pg_info 的 last_backfill 值。如果 pinfo.last_backfill 为 MAX，说明 backfill 操作完成，给该 OSD 发送 MOSDPGBackfill::OP_BACKFILL_FINISH 消息；否则发送 MOSDPGBackfill::OP_BACKFILL_PROGRESS 来更新各个 OSD 上的 pg_info 的 last_backfill 字段。

下面举例说明。

例 11-4　如图 11-4 所示，该 PG 分布在 5 个 OSD 上（也就是 5 个副本，这里为了方便列出各种处理情况），每一行上的对象列表都是相应 OSD 当前对应 backfillInterval 的扫描对象列表。osd5 为主 OSD，是权威的对象列表，其他 OSD 都对照主 OSD 上的对象列表来修复。

osd0	obj4(1,1) last_backfill peer_backfill_info[0].begin	obj5(1,4)	obj6(1,10)
osd1		obj5(1,3) last_backfill peer_backfill_info[1].begin	
osd2	obj4(1,1) last_backfill peer_backfill_info[2].begin	obj5(1,4)	
osd3		obj6(1,1) last_backfill peer_backfill_info[3].begin	obj7(1,8)
osd4		obj5(1,4) last_backfill peer_backfill_info[4].begin	obj6(1,10)
osd5 （主）		**obj5(1,4)** **backfill_info.begin**	**obj6(1,10)**
	last_backfill_started		

图 11-4　backfill 修复过程（1）

下面举例来说明步骤 7 中的不同的修复方法：

1）当前 check 对象指针为主 OSD 上保存的 peer_backfill_info 中 begin 的最小值。图

中 check 对象应为 obj4 对象。

2）比较 check 对象和主 osd5 上的 backfill_info.begin 对象，由于 check 小于 obj5，所以 obj4 为多余的对象，所有拥有该 check 对象的 OSD 都必须删除该对象。故 osd0 和 osd2 上的 obj4 对象被删除，同时对应的 begin 指针前移。

osd0	~~obj4(1,1)~~ last_backfill	obj5(1,4) **peer_backfill_info[0].begin**	obj6(1,10)
osd1		obj5(1,3) last_backfill peer_backfill_info[1].begin	
osd2	~~obj4(1,1)~~ last_backfill	obj6(1,4) **peer_backfill_info[2].begin**	
osd3		obj6(1,1) last_backfill peer_backfill_info[3].begin	obj7(1,8)
osd4		obj5(1,4) last_backfill peer_backfill_info[4].begin	obj6(1,10)
osd5 (pirmary)		**obj5(1,4)** **backfill_info.begin**	**obj6(1,10)**
	last_backfill_started		

图 11-5 backfill 修复过程（2）

3）当前各个 OSD 的状态如图 11-5 所示：此时 check 对象为 obj5，比较 check 和 backfill_info.begin 的值：

a）对于当前 begin 为 check 对象的 osd0、osd1、osd4：

- 对于 osd0 和 osd4，check 对象和 backfill_info.begin 对象都是 obj5，且版本号都为（1,4），加入到 keep_ver_targs 列表中，不需要修复。
- 对于 osd1，版本号不一致，加入 need_ver_targs 列表中，需要修复。

b）对于当前 begin 不是 check 对象的 osd2 和 osd3：

- 对于 osd2，其 last_backfill 小于 backfill_info.begin，显然对象 obj5 缺失，加入 missing_targs 修复。
- 对于 osd3，其 last_backfill 大于 backfill_info.begin，也就是说其已经修复到

obj6 了，obj5 应该已经修复了，加入 skip_targs 跳过。

4）步骤 3 处理完成后，设置 last_backfill_started 为当前的 backfill_info.begin 的值。backfill_info.begin 指针前移，所有 begin 等于 check 对象的 begin 指针前移，重复以上步骤继续修复。

函数 update_range 调用函数 scan_range 更新 BackfillInterval 修复的对象列表，同时检查上次扫描对象列表中，如果有对象发生写操作，就更新该对象修复的版本。

具体实现步骤如下：

1）bi->version 记录了扫描要修复的对象列表时 PG 最新更新的版本号，一般设置为 last_update_applied 或者 info.last_update 的值。初始化时，bi->version 默认设置为（0,0），所以小于 info.log_tail，就更新 bi->version 的设置，调用函数 scan_range 扫描对象。

2）检查如果 bi->version 的值等于 info.last_update，说明从上次扫描对象开始到当前时间，PG 没有写操作，直接返回。

3）如果 bi->version 的值小于 info.last_update，说明 PG 有写操作，需要检查从 bi->version 到 log_head 这段日志中的对象：如果该对象有更新操作，修复时就修复最新的版本；如果该对象已经删除，就不需要修复，在修复队列中删除。

下面举例说明 update_range 的处理过程。

例 11-5　update_range 的处理过程

1）日志记录如下图所示：

obj1(1,2) modify	Obj1(1,3) modify	obj2(1,4) modify	obj3(1,5) modify	obj4(1,6) modify			

扫描列表为：bi->objects　[obj1(1,3), obj2(1,4) obj3(1,5)　obj4(1,6)]

（**begin**）　　（**end**）

BackfillInterval 的扫描的对象列表：bi->begin 为对象 obj1(1,3)，bi->end 为对象 obj6（1,6），当前 info.last_update 为版本（1,6），所以 bi->version 设置为（1,6）。由于本次扫描的对象列表不一定能修复完，只能等下次修复。

2）日志记录如下图所示：

obj1(1,2) modify	Obj1(1,3) modify	obj2(1,4) modify	obj3(1,5) modify	obj4(1,6) modify	obj3(1,7) modify	obj4(1,8) delete	

扫描列表为：bi->objects [obj1(1,3), obj2(1,4) obj3(1,5) ~~obj4(1,6)~~]

(begin) **(end)**

第二次进入函数 recover_backfill，此时 begin 对象指向了 obj2 对象。说明上次只完成了对象 obj1 的修复。继续修复时，期间有对象发生更新操作：

a）对象 obj3 有写操作，版本更新为（1,7）。此时对象列表中要修复的对象 obj3 版本（1,5），需要更新为版本（1,7）的值。

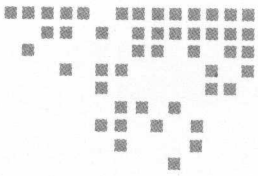
b）对象 obj4 发送删除操作，不需要修复了，所以需要从对象列表中删除。

综上所述可知，Ceph 的 Backfill 过程是扫描 OSD 上该 PG 的所有对象列表，和主 OSD 做对比，修复不存在的或者版本不一致的对象，同时删除多余的对象。

11.5 本章小结

本章介绍了 Ceph 的数据修复的过程，有两个过程：Recovery 过程和 Backfill 过程。Recovery 过程根据 missing 记录，先完成主副本的修复，然后完成从副本的修复。对于不能通过日志修复的 OSD，Backfill 过程通过扫描各个部分上的对象来全量修复。整个 Ceph 的数据修复过程比较清晰，比较复杂的副本可能就是涉及快照对象的修复处理。

目前这部分代码是 Ceph 最核心的代码，除非必要，都不会轻易修改。目前社区也提出了修复时的一种优化方法。就是在日志里记录修改的对象范围，这样在 Recovery 过程中不必拷贝整个对象来修复，只修复修改过的对象对应的范围即可，这样在某些情况下可以减少修复的数据量。



第 12 章 Chapter 12

Ceph 一致性检查

本章介绍 Ceph 的一致性检查工具 Scrub 机制。首先介绍数据校验的基本知识，其次介绍 Scrub 的基本概念，然后介绍 Scrub 的调度机制，最后介绍 Scrub 具体实现的源代码分析。

12.1 端到端的数据校验

在存储系统中可能会发生数据静默损坏（Silent Data Corruption），这种情况的发生大多是由于数据的某一位发生异常反转（Bit Error Rate）。

图 12-1 是一般存储系统的协议栈，数据损坏的情况会发生在系统的所有模块中：

- 硬件错误，例如内存、CPU、网卡等。
- 数据传输过程中的信噪干扰，例如 SATA、FC 等协议。
- 固件 bug，例如 RAID 控制器、磁盘控制、网卡等。
- 软件 bug，例如操作系统内核的 bug，本地文件系统的 bug，SCSI 软件模块的 bug 等。

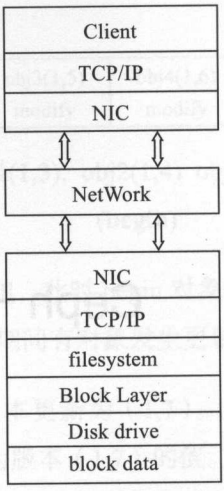


图 12-1　一般存储系统的协议栈

在传统的高端磁盘阵列中，一般采用端到端的数据校验实现数据的一致性。所谓端到端的数据校验，指客户端（应用层）在写入数据时，为每个数据块都计算一个 CRC 校验信息，并将这个校验信息和数据块发送至磁盘（Disk）。磁盘在接收到数据包之后，会重新计算校验信息，并和接收到的校验信息作对比。如果不一致，那么就认为在整个 I/O 路径上存在错误，返回 I/O 操作失败；如果校验成功，就把数据校验信息和数据保存在磁盘上。同样，在数据读取时，客户端再获取数据块和从磁盘读取校验信息时，也需要再次检查是否一致。

通过这种方法，应用层可以很明确地知道一次 I/O 请求的数据是否一致。如果操作成功，那么磁盘上的数据必然是正确的。

这种方式在不影响 I/O 性能或者影响比较小的情况下，可以提高数据读写的完整性。但这种方式也有一些缺点：

- 无法解决目的地址错误导致的数据损坏问题。
- 端到端的解决方案需要在整个 I/O 路径上附加校验信息。现在的 I/O 协议栈涉及的模块比较多，每个模块都附加这种校验信息实现起来比较困难。

由于这种实现方式对 Ceph 的 I/O 性能影响比较大，所以 Ceph 并没有实现端到端的数据校验，而是实现 Ceph Scrub 机制，采用一种通过在后台扫描的方案来解决 Ceph 的一致

性检查。

12.2　Scrub 概念介绍

Ceph 在内部实现了数据一致性检查的一个工具：Ceph Scrub。其原理为：通过对比各个对象副本的数据和元数据，完成副本一致性检查。

这种方法的优点是在后台可以发现由于磁盘损坏而导致的数据不一致现象。缺点是发现的时机往往比较滞后。

Scrub 按照扫描的内容分为两种方式：

- 一种叫 Scrub，它仅仅通过对比对象各副本的元数据，来检查数据的一致性。由于只检查元数据，读取数据量和计算量都比较小，是一种比较轻度的检查。
- 另一种叫 deep-scrub，它进一步检查对象的数据内容是否一致，实现了深度扫描，几乎要扫描磁盘上的所有数据并计算 crc32 校验值，因此比较耗时，占用系统资源更多。

Scrub 按照扫描的方式分为两种：

- 在线扫描：不影响系统正常的业务。
- 离线扫描：需要整个系统暂停或者冻结。

Ceph 的 Scrub 功能实现了在线检查，即不中断系统当前读写请求，客户端可以继续完成读写访问。整个系统并不会暂停，但是后台正在进行 Scrub 的对象要被锁定暂时阻止访问，直到该对象完成 Scrub 操作后才能解锁允许访问。

12.3　Scrub 的调度

Scrub 的调度解决了一个 PG 何时启动 Scrub 扫描机制。主要有以下方式：

- 手动立即启动执行扫描。
- 在后台设置一定的时间间隔，按照间隔的时间来启动。比如默认时间为一天执行一次。

❑ 设置启动的时间段。一般设定一个系统负载比较轻的时间段来执行 Scrub 操作。

12.3.1 相关数据结构

在类 PG 里有下列与 Scrub 相关的数据结构：

```
mutex sched_scrub_lock;          //Scrub 相关变量的保护锁
int scrubs_pending;              // 资源预约已经成功，正等待 Scrub 的 PG
int scrubs_active;               // 正在进行 Scrub 的 PG
set<ScrubJob> sched_scrub_pg;    //PG 对应的所有 ScrubJob 列表
```

结构体 ScrubJob 封装了一个 PG 的 Scrub 任务相关的参数：

```
struct ScrubJob {
  spg_t pgid;              // Scrub 对应的 PG
  utime_t sched_time;      // Scrub 任务的调度时间，如果当前负载比较高，或者当前的时间不在
                              设定的 Scrub 工作时间段内，就会延迟调度
  utime_t deadline;        // 调度时间的上限，过了该时间必须进行 Scrub 操作，而不受系统负载和
                              Scrub 时间段的限制
};
```

12.3.2 Scrub 的调度实现

在 OSD 的初始化函数 OSD::init 中，注册了一个定时任务：

```
tick_timer_without_osd_lock.add_event_after(cct->_conf->osd_heartbeat_
  interval, new C_Tick_WithoutOSDLock(this));
```

该定时任务每隔 osd_heartbeat_interval 时间段（默认为 1 秒），就会触发定时器的回调函数 OSD::tick_without_osd_lock()，处理过程的函数调用关系图如图 12-2 所示。

```
OSD::tick_without_osd_lock
|
—OSD::sched_scrub
|
—PG::sched_scrub
|
—bool PG::queue_scrub
```

图 12-2 Scrub 的触发函数调用关系

以上函数实现了 PG 的 Scrub 调度工作。下面将介绍处理过程中关键的两个函数 OSD::sched_scrub 和 PG::sched_scrub 函数的实现。

1. OSD::sched_scrub 函数

本函数用于控制一个 PG 的 Scrub 过程启动时机，具体过程如下：

1）调用函数 can_inc_scrubs_pending 来检查是否有配额允许 PG 开始启动 Scrub 操作。变量 scrubs_pending 记录了已经完成资源预约正在等待 Scrub 的 PG 的数量，变量 scrubs_active 记录了正在进行 Scrub 检查的 PG 数量，其二者的数量之和不能超过系统配置参数 cct->_conf->osd_max_scrubs 的值。该值设置了同时允许 Scrub 的最大的 PG 数量。

2）调用函数 scrub_time_permit 检查是否在允许的时间段内。如果 cct->_conf->osd_scrub_begin_hour 大于 cct->_conf->osd_scrub_end_hour，当前时间必须在二者设定的时间范围之间才允许。如果 cct->_conf->osd_scrub_begin_hour 小于等于 cct->_conf->osd_scrub_end_hour，当前时间在二者设定的时间范围之外才允许。

3）调用函数 scrub_load_below_threshold 检查当前的系统负载是否允许。函数 getloadavg 获取最近 1、5、15 分钟的系统负载。

a）如果最近 1 分钟的负载小于 cct->_conf->osd_scrub_load_threshold 的设定值，就允许执行。

b）如果最近 1 分钟的负载小于 daily_loadavg 的值，并且最近 1 分钟负载小于最近 15 分钟的负载，就允许执行。

4）获取第一个等待执行 Scrub 操作的 ScrubJob 列表，如果它的 scrub.sched_time 大于当前时间 now 值，说明时间不到，就跳过该 PG 先执行下一个任务。

5）获取该 PG 对应的 PG 对象，如果该 PG 的 pgbackend 支持 Scrub，并且处于 active 状态：

a）如果 scrub.deadline 小于 now 值，也就是已经超过最后的期限，必须启动 Scrub 操作。

b）或者此时 time_permit 并且 load_is_low，也就是时间和负载都允许。

在上述两种情况下，调用函数 pg->sched_scrub() 来执行 Scrub 操作。

2. PG::sched_scrub 函数

本函数实现了对执行 Scrub 任务时相关参数的设置，并完成了所需资源的预约。其处理过程如下：

1）首先检查 PG 的状态，必须是主 OSD，并且处于 active 和 clean 状态，并且没有正在进行 Scrub 操作。

2）设置 deep_scrub_interval 值：如果该 PG 所在的 pool 选项中没有设置该值，就设置为系统配置参数 cct->_conf->osd_deep_scrub_interval 的值。

3）检查是否启动 deep_scrub，如果当前时间大于 info.history.last_deep_scrub_stamp 与 deep_scrub_interval 之和，就启动 deep_scrub 操作。

4）如果 scrubber.must_scrub 的值为 true，为用户手动强制启动 deep_scrub 操作。如果该值为 false，则需系统自动以一定的概率来启动 deep_scrub 操作，具体实现就是：自动产生一个随机数，如果该随机数小于 cct->_conf->osd_deep_scrub_randomize_ratio，就启动 deep_scrub 操作。

5）决定最终是否启动 deep_scrub，在步骤 3）和 4）中只要有一个设置好，就启动 deep_scrub 操作。

6）如果 osdmap 或者 pool 中带有不支持 deep_scrub 的标记，就设置 time_for_deep 为 false，不启动 deep_scrub 操作。

7）如果 osdmap 或者 pool 中带有不支持 Scrub 的标记，并且也没有启动 deep-scrub 操作则返回并退出。

8）如果 cct->_conf->osd_scrub_auto_repair 设置了自动修复，并且 pgbackend 也支持，而且是 deep_scrub 操作，则进行如下判断过程：

a）如果用户设置了 must_repair，或者 must_scrub，或者 must_deep_scrub，说明这次 Scrub 操作是用户触发的，系统尊重用户的选择，不会自动设置 scrubber.auto_repair 的值为 true。

b）否则，系统就设置 scrubber.auto_repair 的值为 true 来自动进行修复。

9）Scrub 过程和 Recovery 过程类似，都需要耗费系统大量资源，需要去 PG 所在的 OSD 上预约资源。如果 scrubber.reserved 的值为 false，还没有完成资源的预约，需进行如下操作：

a）把自己加入到 scrubber.reserved_peers 中。

b）调用函数 scrub_reserve_replicas，向 OSD 发送 CEPH_OSD_OP_SCRUB_RESERVE 消息来预约资源。

c）如果 scrubber.reserved_peers.size() 等于 acting.size()，表明所有的从 OSD 资源预约成功，把 PG 设置为 PG_STATE_DEEP_SCRUB 状态。调用函数 queue_

scrub 把该 PG 加入到工作队列 op_wq 中，触发 Scrub 任务开始执行。

12.4　Scrub 的执行

Scrub 的具体执行过程大致如下：通过比较对象各个 OSD 上副本的元数据和数据，来完成元数据和数据的校验。其核心处理流程在函数 PG::chunky_scrub 里控制完成。

12.4.1　相关数据结构

Scrub 操作相关的主要数据结构有两个，一个是 Scrubber 控制结构，它相当于是一次 Scrub 操作的上下文，控制一次 PG 的操作过程。另一个是 ScurbMap 保存需要比较的对象各个副本的元数据和数据的摘要信息。

1. Scrubber

结构体 Scrubber 用来控制一个 PG 的 Scrub 的过程：

```
struct Scrubber {
  // 元数据
  set<pg_shard_t> reserved_peers;              // 资源预约的 shard
  bool reserved, reserve_failed;               // 是否预约资源，预约资源是否失败
  epoch_t epoch_start;                         // 开始 Scrub 操作的 epoch

  // common to both scrubs
  bool active;                                 //Scrub 是否开始
  bool queue_snap_trim;
  // 当 PG 有 snap_trim 操作时，如果检查 Scrubber 处于 active 状态，说明正在进行 Scrub 操作，那
     么 snap_trim 操作暂停，设置 queue_snap_trim 的值为 true。当 PG 完成 Scrub 任务后，如果
     queue_snap_trim 的值为 true，就把 PG 添加到相应的工作队列里，继续完成 snap_trim 操作。

  int waiting_on;                              // 等待的副本计数
  set<pg_shard_t> waiting_on_whom;             // 等待的副本

  int shallow_errors;                          // 轻度扫描错误数
  int deep_errors;                             // 深度扫描错误数
  int fixed;                                   // 已经修复的对象数

  ScrubMap primary_scrubmap;                   // 主副本的 ScrubMap
  map<pg_shard_t, ScrubMap> received_maps;     // 接收到的从副本的 ScrubMap
  OpRequestRef active_rep_scrub;
```

```
  utime_t scrub_reg_stamp;                    // stamp we registered for
  // flags to indicate explicitly requested scrubs (by admin)
  bool must_scrub, must_deep_scrub, must_repair;

     bool auto_repair;                        //是否自动修复

  // Maps from objects with errors to missing/inconsistent peers
  map<hobject_t, set<pg_shard_t>, hobject_t::BitwiseComparator> missing;
  //扫描出的缺失对象
  map<hobject_t, set<pg_shard_t>, hobject_t::BitwiseComparator> inconsistent;
                                              //扫描出的不一致对象

  // Map from object with errors to good peers
  map<hobject_t, list<pair<ScrubMap::object, pg_shard_t> >, hobject_
  t::BitwiseComparator> authoritative;
//如果所有副本对象中有不一致的对象，authoritative 记录了正确对象所在的 OSD
  // digest updates which we are waiting on
  int num_digest_updates_pending;             //等待更新 digest 的对象数目

  hobject_t start, end;                       //扫描对象列表的开始和结尾
  eversion_t subset_last_update;              //扫描对象列表中最新的版本号

  bool deep;                                  //是否为深度扫描
  uint32_t seed;                                  //计算 crc32 校验码的种子

  list<Context*> callbacks;

} scrubber;
```

2. ScrubMap

数据结构 ScrubMap 保存准备校验的对象以及相应的校验信息：

```
struct ScrubMap {
  struct object {
    map<string,bufferptr> attrs;              //对象的属性
    set<snapid_t> snapcolls;                  //该对象所有的 snap 序号
    uint64_t size;                            //对象的 size
    __u32 omap_digest;                        //omap 的 crc32c 校验码
    __u32 digest;                             //对象数据的 crc32c 校验码
    uint32_t nlinks;                          //snap 对象（clone 对象）对应的 snap 的数量
    bool negative:1;
    bool digest_present:1;                    //是否计算了数据的校验码标志
    bool omap_digest_present:1;               //是否有 omap 的校验码标志
    bool read_error:1;                        //读对象的数据出错标志
    bool stat_error:1;                        //调用 stat 获取对象的元数据出错标志
  };
```

```
    ……
    map<hobject_t,object, hobject_t::BitwiseComparator> objects;
        // 需要校验的对象（hobject）→校验信息（Object）的映射
    eversion_t valid_through;
    eversion_t incr_since;
    ……
};
```

内部类 object 用来保存对象需要校验的信息，包括以下 5 个方面：

- 对象的大小（size）
- 对象的属性（attrs）
- 对象 omap 的校验码（digest）
- 对象数据的校验码（digest）
- 对象所有 clone 对象的快照序号

12.4.2　Scrub 的控制流程

Scrub 的任务由 OSD 的工作队列 OpWq 来完成，调用对应的处理函数 pg->scrub(handle) 来执行。

PG::scrub 函数最终调用 PG::chunky_scrub 函数来实现。PG::chunky_scrub 函数控制了 Scrub 操作状态转换和核心处理过程。

具体过程分析如下所示：

1）Scurbber 的初始状态为 PG::Scrubber::INACTIVE，该状态的处理如下：

a）设置 scrubber.epoch_start 的值为 info.history.same_interval_since。

b）设置 scrubber.active 的值为 true。

c）设置状态 scrubber.state 的值为 PG::Scrubber::NEW_CHUNK。

d）根据 peer_features，设置 scrubber.seed 的类型，这个 seed 是计算 crc32 的初始化哈希值。

2）PG::Scrubber::NEW_CHUNK 状态的处理如下：

a）调用 get_pgbackend()->objects_list_partial 函数从 start 对象开始扫描一组对象，一次扫描的对象数量在如下两个配置参数之间：cct->_conf->osd_scrub_chunk_

min（默认值为 5）和 cct->_conf->osd_scrub_chunk_max（默认值为 25）。

b）计算出对象的边界。相同的对象具有相同的哈希值。从列表后面开始查找哈希值不同的对象，从该地方划界。这样做的目的是把一个对象的所有相关对象（快照对象、回滚对象）划分在一次扫描校验过程中。

c）调用函数 _range_available_for_scrub 检查列表中的对象，如果有被阻塞的对象，就设置 done 的值为 true，退出 PG 本次的 Scrub 过程。

d）根据 pg_log 计算 start 到 end 区间对象最大的更新版本号，这个最新版本号设置在 scrubber.subset_last_update 里。

e）调用函数 _request_scrub_map 向所有副本发送消息，获取相应 ScrubMap 的校验信息。

f）设置状态为 PG::Scrubber::WAIT_PUSHES。

3）PG::Scrubber::WAIT_PUSHES 状态的处理如下：

a）如果 active_pushes 的值为 0，设置状态为 PG::Scrubber::WAIT_LAST_UPDATE，进入下一个状态处理。

b）如果 active_pushes 不为 0，说明该 PG 正在进行 Recovery 操作。设置 done 的值为 true，直接结束。在进入 chunky_scrub 时，PG 应该处于 CLEAN 状态，不可能有 Recovery 操作，这里的 Recovery 操作可能是上次进行 chunky_scrub 操作后的修复操作。

4）PG::Scrubber::WAIT_LAST_UPDATE 状态的处理如下：

a）如果 last_update_applied 的值小于 scrubber.subset_last_update 的值，说明虽然已经把操作写入了日志，但是还没有应用到对象中。由于 Scrub 操作后面的步骤有对象的读操作，所以需要等待日志应用完成。设置 done 的值为 true 结束本次 PG 的 Scrub 过程。

b）否则就设置状态为 PG::Scrubber::BUILD_MAP。

5）PG::Scrubber::BUILD_MAP 状态的处理如下：

a）调用函数 build_scrub_map_chunk 构造主 OSD 上对象的 ScrubMap 结构。

b）如果构造成功，计数 scrubber.waiting_on 的值减 1，并从队列中删除 scrubber.

waiting_on_whom，则相应的状态设置为 PG::Scrubber::WAIT_REPLICAS。

6）PG::Scrubber::WAIT_REPLICAS 状态的处理如下：

a）如果 scrubber.waiting_on 不为零，说明有 replica 请求没有回答，设置 done 的值为 true，退出并等待。

b）否则，进入 PG::Scrubber::COMPARE_MAPS 状态。

7）PG::Scrubber::COMPARE_MAPS 状态的处理如下：

a）调用函数 scrub_compare_maps 比较各副本的校验信息。

b）将参数 scrubber.start 的值更新为 scrubber.end。

c）调用函数 requeue_ops，把由于 Scrub 而阻塞的读写操作重新加入操作队列里执行。

d）状态设置为 PG::Scrubber::WAIT_DIGEST_UPDATES。

8）PG::Scrubber::WAIT_DIGEST_UPDATES 状态的处理如下：

a）如果有 scrubber.num_digest_updates_pending 等待，等待更新数据的 digest 或者 omap 的 digest。

b）如果 scrubber.end 小于 hobject_t::get_max()，本 PG 还有没有 Scrub 操作完成的对象，设置状态 scrubber.state 为 PG::Scrubber::NEW_CHUNK，继续把 PG 加入到 osd->scrub_wq 中。

c）否则，设置状态为 PG::Scrubber::FINISH 值。

9）PG::Scrubber::FINISH 状态的处理如下：

a）调用函数 scrub_finish 来设置相关的统计信息，并触发修复不一致的对象。

b）设置状态为 PG::Scrubber::INACTIVE。

12.4.3　构建 ScrubMap

构建 ScrubMap 有多个函数实现，下面分别介绍。

1. build_scrub_map_chunk

函数 build_scrub_map_chunk 用于构建从 start 到 end 之间的所有对象的校验信息并保

存在 ScrubMap 结构中。

```
int PG::build_scrub_map_chunk(
  ScrubMap &map,
  hobject_t start, hobject_t end,
  bool deep, uint32_t seed,
  ThreadPool::TPHandle &handle)
```

处理过程分析如下：

1）设置 map.valid_through 的值为 info.last_update。

2）调用 get_pgbackend()->objects_list_range 函数列出所有的 start 和 end 范围内的对象，ls 队列存放 head 和 snap 对象，rollback_obs 队列存放用来回滚的 ghobject_t 对象。

3）调用函数 get_pgbackend()->be_scan_list 扫描对象，构建 ScrubMap 结构。

4）调用函数 _scan_rollback_obs 来检查回滚对象：如果对象的 generation 小于 last_rollback_info_trimmed_to_applied 值，就删除该对象。

5）调用 _scan_snaps 来修复 SnapMapper 里保存的 snap 信息。

2. _scan_snaps

函数 _scan_snaps 扫描 head 对象保存的 snap 信息和 SnapMapper 中保存的该对象的 snap 信息是否一致。它以前者保存的对象 snap 信息为准，修复 snapMapper 中保存的对象 snap 信息。

```
void PG::_scan_snaps(ScrubMap &smap)
```

具体实现过程为：对于 ScrubMap 里的每一个对象循环做如下操作：

1）如果对象的 hoid.snap 的值小于 CEPH_MAXSNAP 的值，那么该对象是 snap 对象，从 o.attrs[OI_ATTR] 里获取 object_info_t 信息。

2）检查 oi 的 snaps。如果 oi.snaps.empty() 为 0，设置 nlinks 等于 1；如果 oi.snaps.size() 为 1，设置 nlinks 等于 2；否则设置 nlinks 等于 3。

3）从 oi 获取 oi_snaps，从 snap_mapper 获取 cur_snaps，比较两个 snap 信息，以 oi 的信息为准：

a）如果函数 snap_mapper.get_snaps(hoid, &cur_snaps) 的结果为 -ENOENT，就把信息该添加到 snap_mapper 里。

b）如果信息不一致，先删除 snap_mapper 里不一致的对象，然后把该对象的 snap 信息添加到 snap_mapper 里。

3. be_scan_list

函数 be_scan_list 用于构建 ScubMap 中对象的校验信息：

```
void PGBackend::be_scan_list(
  ScrubMap &map,
  const vector<hobject_t> &ls,
  bool deep,
  uint32_t seed,
  ThreadPool::TPHandle &handle)
```

具体处理过程就是循环扫描 ls 向量中的对象：

1）调用 store->stat 获取对象的 stat 信息：

a）如果获取成功，设置 o.size 的值等于 st.st_size，并调用 store->getattrs 把对象的 attr 信息保存在 o.attrs 里。

b）如果 stat 返回结果 r 为 -ENOENT，就直接跳过该对象（该对象在本 OSD 上可能缺失，在后面比较结果时会检查出来）。

c）如果 stat 返回结果 r 为 -EIO，就设置 o.stat_error 的值为 true。

2）如果 deep 的值为 true，调用函数 be_deep_scrub 进行深度扫描，获取对象的 omap 和 data 的 digest 信息。

4. be_deep_scrub

函数 be_deep_scrub 实现对象的深度扫描：

```
void ReplicatedBackend::be_deep_scrub(
  const hobject_t &poid,          // 深度扫描的对象
  uint32_t seed,                  //crc32 的种子
  ScrubMap::object &o,            // 保存对应的校验信息
  ThreadPool::TPHandle &handle)
```

实现过程分析如下：

1）设置 data 和 omap 的 bufferhash 的初始值都为 seed。

2）循环调用函数 store->read 读取对象的数据，每次读取长度为配置参数 cct->_conf->osd_deep_scrub_stride（512k），并通过 bufferhash 计数 crc32 校验值。如果中间出错（r == -EIO），就设置 o.read_error 的值为 true。最后设置 o.digest 为计算出 crc32 的校验值，设置 o.digest_present 的值为 true。

3）调用函数 store->omap_get_header 获取 header，迭代获取对象的 omap 的 key-value 值。计算 header 和 kev-value 的 digest 信息，并设置在 o.omap_digest 中，标记 o.omap_digest_present 的值为 true。

综上可知，通过函数 be_scan_list 来获取对象的元数据信息，通过 be_deep_scrub 函数获取对象的数据和 omap 的 digest 信息保存在 ScrubMap 结构中。

12.4.4 从副本处理

当从副本接收到主副本发送来的 MOSDRepScrub 类型消息，用于获取对象的校验信息时，就调用函数 replica_scrub 来完成。

函数 replica_scrub 具体实现如下：

1）首先确保 scrubber.active_rep_scrub 不为空。

2）检查如果 msg->map_epoch 的值小于 info.history.same_interval_since 的值就直接返回。在这里从副本直接丢弃掉过时的 MOSDRepScrub 请求。

3）如果 last_update_applied 的值小于 msg->scrub_to 的值，也就是从副本上完成日志应用的操作落后主副本 scrub 操作的版本，必须等待它们一致。把当前的 op 操作保存在 scrubber.active_rep_scrub 中等待。

4）如果 active_pushes 大于 0，表明有 Recovery 操作正在进行，同样把当前的 op 操作保存在 scrubber.active_rep_scrub 中等待。

5）否则就调用函数 build_scrub_map_chunk 来构建 ScrubMap，并发送给主副本。

当等待的本地操作应用完成之后，在函数 ReplicatedPG::op_applied 检查，如果 scrubber.active_rep_scrub 不为空，并且该操作的版本等于 msg->scrub_to，就会把保存的 op 操作重新放入 osd->op_wq 请求队列，继续完成该请求。

12.4.5　副本对比

当对象的主副本和从副本都完成了校验信息的构建，并保存在相应的结构 ScrubMap 中，下一步就是对比各个副本的校验信息来完成一致性检查。首先通过对象自身的信息来选出一个权威的对象，然后用权威对象和其他对象做比较来检验。下面介绍用于比较的函数。

1. scrub_compare_maps

函数 scrub_compare_maps 实现比较不同的副本信息是否一致，处理过程如下：

1）首先确保 acting.size() 大于 1，如果该 PG 只有一个 OSD，则无法比较。

2）把 actingbackfill 对应 OSD 的 ScrubMap 放置到 maps 中。

3）调用函数 be_compare_scrubmaps 来比较各个副本的对象，并把对象完整的副本所在 shard 保存在 authoritative 中。

4）调用 _scrub 函数继续比较 snap 之间对象的一致性。

2. be_compare_scrubmaps

函数 be_compare_scrubmaps 用来比较对象各个副本的一致性，其具体处理过程分析如下：

1）首先构建 master set，也就是所有副本 OSD 上对象的并集。

2）对 master set 中的每一个对象进行如下操作：

a）调用函数 be_select_auth_object 选择出一个具有权威对象的副本 auth，如果没有选择出权威对象，变量 shallow_errors 加一来记录这种错误。

b）调用函数 be_compare_scrub_objects，比较各个 shard 上的对象和权威对象：分别对 data 的 digest、omap 的 omap_digest 和属性 attrs 进行对比：

- 如果结果为 clean，表明该对象和权威对象的各项比较完全一致，就把该 shard 添加到 auth_list 列表中。
- 如果结果不为 clean，就把该对象添加到 cur_inconsistent 列表中，分别统计 shallow_errors 和 deep_errors 的值。

- 如果该对象在该 shard 上不存在，添加到 cur_missing 列表中，统计 shallow_errors 值。

c）检查该对象所有的比较结果：如果 cur_missing 不为空，就添加到 missing 队列列；如果有 cur_inconsistent 对象，就添加到 inconsistent 对象里；如果该对象有不完整的副本，就把没有问题的记录放在 authoritative 中。

d）如果权威对象 object_info 里记录的 data 的 digest 和 omap 的 omap_digest 和实际扫描出数据计算的结果不一致，update 的模式就设置为 FORCE，强制修复。如果 object_info 里没有 data 的 digest 和 omap 的 digest，修复的模式 update 设置为 MAYBE。

e）最后检查，如果 update 的模式为 FORCE，或者该对象存在的时间 age 大于配置参数 g_conf->osd_deep_scrub_update_digest_min_age 的值，就加入 missing_digest 列表中。

3. be_select_auth_object

函数 be_select_auth_object 用于在各个 OSD 上的副本对象中，选择出一个权威的对象：auth_obj 对象。其原理是根据自身所带的冗余信息来验证自己是否完整，具体流程如下：

1）首先确认该对象的 read_error 和 stat_error 没有设置，也就在获取对象的数据和元数据的过程中没有出错，否则就直接跳过。

2）确认获取的属性 OI_ATTR 值不为空，并将数据结构 object_info_t 正确解码，就设置当前的对象为 auth_obj 对象。

3）验证保存在 objec_info_t 中的 size 值和扫描获取对象的 size 值是否一致，如果不一致，就继续查找一个更好的 auth_obj 对象。

4）如果是 replicated 类型的 PG，验证在 object_info_t 里保存的 data 和 omap 的 digest 值是否和扫描过程计算出的值一致。如果不一致，就继续查找一个更好的 auth_obj 对象。

5）如上述都一致，直接终止循环，当前已经找到满意的 auth_obj 对象。

由上述的选择过程可知，选中一个权威对象的条件如下：

- 步骤 1），2）两点条件是基础，对象的数据和属性能正确读取。
- 步骤 3），4）是利用了 object_info_t 中保存的对象 size，以及 omap 和 data 的 digest

的冗余信息。用这些信息和对象扫描读取的数据计算出的信息比较来验证。

4. _scrub

函数 _scrub 检查对象和快照对象之间的一致性：

```
void ReplicatedPG::_scrub(ScrubMap &scrubmap,
  const map<hobject_t, pair<uint32_t, uint32_t> > &missing_digest)
```

如果 pool 有 cache pool 层，那么就允许拷贝对象有不一致的状态，因为有些对象可能还存在于 cache pool 中没有被刷回。通过函数 pool.info.allow_incomplete_clones() 来确定上述情况。

其实代码比较复杂，下面通过举例来说明其实现基本过程。

例 12-1　_scrub 函数实现过程举例，如表 12-1 所示。

表 12-1　_scrub 函数实现过程

步骤	对象列表	希望的对象
1	obj1 snap1	head/snapdir, unexpected obj1 snap 1
2	obj2 head	head/snapdir, head ok [snapset clones 6 4 2 1]
3	obj2 snap7	obj2 snap 6, unexpected obj2 snap 7
4	obj2 snap6	obj2 snap 6, match
5	obj2 snap4	obj2 snap 4, match
6	obj3 head	obj2 snap 2 (expected), obj2 snap 1 (expected), head ok [snapset clones 3 1]
7	obj3 snap3	obj3 snap 3 match
8	obj3 snap1	obj3 snap 1 match
9	obj4 snapdir	head/snapdir, snapdir ok [snapset clones 4]
10	EOL	obj4 snap 4, (expected)

_scurb 实现过程说明如下：

1）对象 obj1 snap1 为快照对象，就应该有 head 或者 snapdir 对象。但是该快照对象没有对应的 head 或者 snapdir 对象，那么该对象被标记为 unexpected 对象。

2）对象 obj2 head 是一个 head 对象，这是预期的对象。通过 head 对象，获取 snapset

的 clones 列表为 [6,4,2,1]。

3）检查对象 obj2 snap7 并没有在对象 obj2 的 snapset 的 clones 列表中，为异常对象。

4）对象 obj2 的快照对象 snap6，在 snapset 的 clones 列表中。

5）对象 obj2 的快照对象 snap4，在 snapset 的 clones 列表中。

6）当遇到 obj3 head 对象时，预期的对象 obj2 中应该还有快照对象 snap2 和 snap1 为缺失的对象。继续获取 snap3 的 snapset 的 clones 值为列表 [3,1]。

7）对象 obj3 的快照对象 snap3 和预期对象一致

8）对象 obj3 的快照对象 snap1 和预期对象一致。

9）对象 obj4 的 snapdir 对象，符合预期。获取该对象的 snapset 的 clones 值为列表 [4]。

10）扫描的对象列表结束了，但是预期对象为 obj4 的快照对象 snap4，对象 obj4 snap4 缺失。

目前对于 excepted 的对象和 unexcepted 的对象，都只是在日志中标记出来，并不做进一步的处理。

最后对于那些其他都正确但对象的 digest 不正确的数据对象，也就是 missing_digest 中需要更新 digest 的对象，发送更新 digest 请求。

12.4.6 结束 Scrub 过程

scrub_finish 函数用于结束 Scrub 过程，其处理过程如下：

1）设置了相关 PG 的状态和统计信息。

2）调用函数 scrub_process_inconsistent 用于修复 scrubber 里标记的 missing 和 inconsistent 对象，其最终调用 repair_object 函数。它只是在 peer_missing 中标记对象缺失。

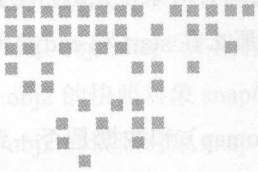
3）最后触发 DoRecovery 事件发送给 PG 的状态机，发起实际的对象修复操作。

12.5 本章小结

本章介绍了 Scrub 的基本原理，及 Scrub 过程的调度机制，然后介绍了校验信息的构建和比较验证的具体流程。

最后，总结一下 Ceph 一致性检查 Scrub 功能实现的关键点：

- 如果做 Scrub 操作检查的对象范围 start 和 end 之间有对象正在进行写操作，就退出本次 Scrub 进程；如果已经开始启动了 Scrub，那么在 start 和 end 之间的对象写操作需要等待 Scrub 操作的完成。
- 检查就是对比主从副本的元数据（size、attrs、omap）和数据是否一致。权威对象的选择是根据对象自己保持的信息（object info）和读取对象的信息是否一致来选举出。然后用权威对象对比其他对象副本来检查。



Chapter 13

第 13 章

Ceph 自动分层存储

本章介绍 Ceph 的 Cache Tier 模块，它实现了的自动分层存储技术。本章首先介绍自动分层存储的概念和原理，其次介绍 Ceph 的自动分层存储 Cache Tier 的读写模式，最后对 Cache Tier 的源代码进行详细的分析。

13.1 自动分层存储技术

分层存储在“信息生命周期管理”的基础上对数据进一步分层。这个概念的提出基于这样一个事实：在数据的不同生命周期里，其访问的频度是截然不同的，即使是处于同一生命周期的不同类型的数据，访问频率也会不同。

自动分层存储技术目标在于：把用户访问频率高的数据放置在高性能、小容量的存储介质中，把大量低频访问的数据放置在大容量、性能相对较低的存储介质中。它为用户提供的价值在于：在提高热点数据存储性能的同时，降低了存储成本。首先，数据可以自由安全地迁移到更低层的存储介质中，这样可以节约存储成本；其次热点数据可以自动的从低层迁移到高层存储层，提高访问热点数据的性能。

自动分层存储技术实现的核心点在于：

- 数据访问行为的追踪、统计与分析：持续追踪与统计每个数据块的存取频率，并通过定期分析，识别出存取频率高的“热”区块，与存取频率低的“冷”区块。
- 数据迁移：以存取频率为基础，定期执行数据迁移，将热点数据块迁移到高速存储层，把较不活跃的冷数据块迁移到低速存储层。

传统的磁盘阵列存储产品的自动分层存储技术中，数据访问行为统计和分析的粒度是以数据块为单位的。在 Ceph 中，数据访问行为统计、分析以及数据在各存储层之间迁移的粒度是以对象（默认为 4MB）为基本单位的。

13.2　Ceph 分层存储架构和原理

在 Ceph 里，Cache Tier 模块在 pool 层设置。可以设置一个 pool 为另一个 pool 的 cache 层。在这里，做缓存层的 pool 称为 cache pool（或者 cache tier、hot pool 等）；相对低性能的存储层称为 data pool（或者 base pool、cold pool 等）。

图 13-1 是 Cache Tier 的存储架构图。在 client 层的 librados 对于 Cache Tier 是透明的，它对 Cache Tier 是无感知的。Objecter 实现了 Cache Tier 相关的逻辑处理。Cache Tier 层为高速存储层，热点数据都保存在 Cache Tier 层，Data Tier 层为慢速设备存储层，所有数据最终都会保存在 Data Tier 层。

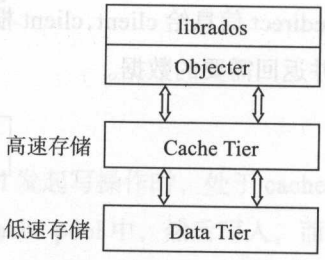


图 13-1　Cache Tier 存储架构图

13.3　Cache Tier 的模式

在结构 cache_mode_t 的枚举类型里，定义 cache pool 有几种模式：

- write back
- read forward
- read proxy
- write proxy
- read only

下面将详细介绍各种模式的不同应用场景。

1. write back 模式

在 write back 模式下，读写请求都直接发给 cache pool，这种模式适合于大量修改数据的应用场景（例如图片视频编辑、事务处理 OLTP 类应用）。

在 write back 模式下，读操作时对象在缓存层不存在（cache miss）时，有 read forward 和 read proxy 两种特殊模式进行处理。写操作在缓存不命中的情况下有 writeproxy 的特殊模式进行处理。

2. read forward 模式

当进行 read 操作时，对象不在 cache pool 中，就直接返回，client 直接向 data pool 发送请求，数据直接返回给 client。

如图 13-2 所示，当客户端向 cache pool 发起读请求时，出现 cache miss 状态，就返回 redirect 信息给 client，client 根据返回的 redirect 信息，再次直接向 base pool 层发起读请求，并返回需要的数据。

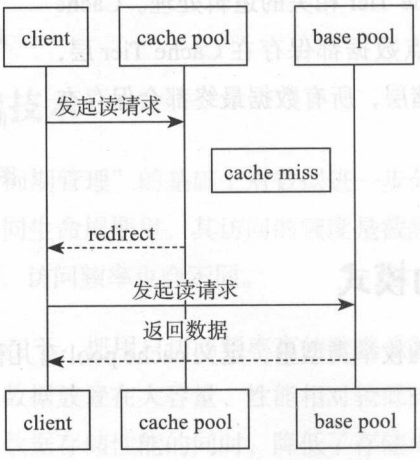


图 13-2 read forward 模式

3. Read proxy 模式

当进行 read 操作时，对象不在 cache pool 中，cache pool 层向 data pool 发送请求，返

回给 cache pool 层，cache pool 层再返回给 client 数据。

如图 13-3 所示，当 client 向 cache pool 发起读请求，该对象在 cache pool 处于 cache miss 状态，cache pool 层向 base pool 层发起读请求，返回数据给 cache pool 层，cache pool 层再将数据返回给 client。

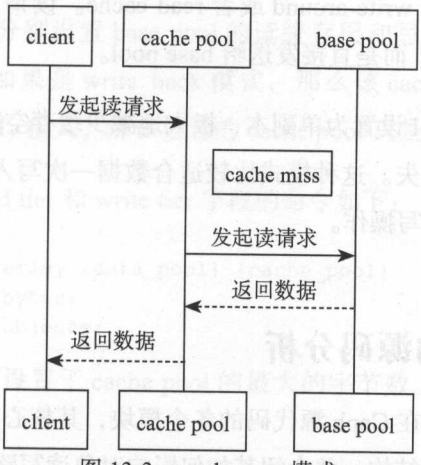


图 13-3　read proxy 模式

4. write proxy 模式

图 13-4 是 write proxy 模式示意图，当客户端先 cache pool 发起写操作时，处于 cache miss 的状态，cache pool 并不等该对象从 base pool 中加载到 cache pool 中，然后写入，而是直接发请求给 base pool，直接把数据写入 base pool 层。

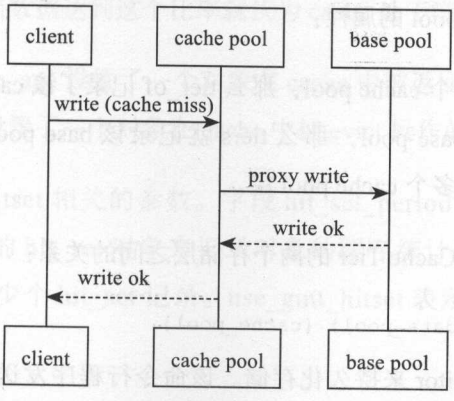


图 13-4　write proxy 模式

write proxy 模式中，当 cache pool 处于 cache miss 状态时，直接写入 base pool，降低了写操作的延迟，提高了写操作的性能。

5. read only 模式

read only 模式也称为：write-around 或者 read cache。读请求直接发送给 cache pool，写请求并不经过 cache pool，而是直接发送给 base pool。

这种方式下，cache pool 设置为单副本，极大地减少缓存空间的占用库。当 cache pool 层失效时，也不会有数据丢失。这种模式比较适合数据一次写入多次读取的应用场景。例如图片、视频、音频等的读写操作。

13.4 Cache Tier 的源码分析

Cache Tier 的代码分布在 Ceph 源代码的各个模块，其核心在对象的数据读写路径上。下面就着重分析相关的数据结构，并介绍其如何影响对象读写路径的。

13.4.1 pool 中的 Cache Tier 数据结构

数据结构 pg_pool_t 里有 Cache Tier 相关数据字段，如下所示：

```
set<uint64_t> tiers;
int64_t tier_of;
```

这两个字段用来设置 pool 的属性：

- 如果当前 pool 是一个 cache pool，那么 tier_of 记录了该 cache pool 的 base pool 层。
- 如果当前 pool 是 base pool，那么 tiers 就记录该 base pool 的 cache pool 层，一个 base pool 可以设置多个 cache pool 层。

使用如下命令来设置 Cache Tier 的两个存储层之间的关系：

```
ceph osd tier add {data_pool} {cache_pool}
```

pool 的信息都由 Monitor 来持久化存储。该命令行程序发送请求给 Monitor，然后由 Monitor 相关 pool 设置上述属性值：

```
cache_mode_t cache_mode;  // cache 的模式
```

cache_mode 设置了 cache 的模式，可以通过下面命令设置：

```
ceph osd tier cache-mode {cachepool} {cache-mode}
  int64_t read_tier;
  int64_t write_tier;
```

read_tier 和 write_tier 分别设置 base pool 的读缓存层和写缓存层。根据 Ceph 不同的 Cache Tier 模式来设置。如果是 write_back 模式，那么该 cache pool 既是 read 层，又是 write 层。如果只是 read only 模式，那么设置的 cache pool 只是 read 层，并不是 write 层。

根据模式分别设置 read tier 和 write tier 字段的命令如下：

```
ceph osd tier set-overlay {data_pool} {cache_pool}
uint64_t target_max_bytes;
uint64_t target_max_objects;
```

字段 target_max_bytes 设置了 cache pool 的最大的字节数。target_max_objects 设置了 cache pool 的最大对象数量。

```
uint32_t cache_target_dirty_ratio_micro;
uint32_t cache_target_dirty_high_ratio_micro;
uint32_t cache_target_full_ratio_micro;
```

上面这三项分别为：

- 目标脏数据率：当脏数据率达到这个值时，后台 agent 开始 flush 数据
- 高目标脏数据率：当脏数据率达到这个值时，后台 agent 开始高速 flush 数据
- 数据满的比率：当数据达到这个比率就认为 cache 处于满的状态

字段 cache_min_flush_age 设置了一个对象在 cache 中被返回 base tier 的最小时间。字段 cache_min_evict_age 设置了一个对象在 cache 中被 evict 操作的最小操作时间。

hit_set_params 是 hitset 相关的参数。字段 hit_set_period 用来设置每过该段时间，系统要重新产生一个新的 hit_set 对象来记录对象的缓存统计信息。hit_set_count 用来记录系统保存最近的多少个 hit_set 记录。use_gmt_hitset 表示 hitset archive 对象的命名规则。

13.4.2 HitSet

类 HitSet 用来跟踪和统计对象的访问行为。目前仅记录了对象是否在缓存中。它定义了一个缓存查找的抽象接口。目前的三种实现方式分别为：ExplicitObjectHitSet、ExplicitHashHitSet、BloomHitSet。下面分别介绍

1. ExplicitObjectHitSet

类 ExplicitObjectHitSet 用一个基于 hobject 的 set 来记录对象的命中：

```
ceph::unordered_set<hobject_t> hits;
```

这种方式实现简单，直观。但是缺点也很明显，要在内存中缓存数据结构 hobject，其占用内存空间比较大。

可以粗略地统计一下，在假设对象的名字占 40 个字节，一个 hobject 大约占 100 字节那么大：

400G 的 SSD 盘（卡）的占用空间	400G / 4M × 100 = 10M
4T 的 SSD 盘（卡）的占用空间	4T / 4M × 100 = 100M

2. ExplicitHashHitSet

类 ExplicitHashHitSet 基于对象 32 位哈希值的 set 来记录的对象命中：

```
ceph::unordered_set<uint32_t> hits;
```

每个对象占用 4 字节的内存空间，占用内存空间如下：

400G 的 SSD 盘（卡）的占用空间	400G / 4M × 4 = 400k
4T 的 SSD 盘（卡）的占用空间	4T / 4M × 4 = 4M

这种方式占用空间相对较少。但缺点也比较明显：当知道一个对象的哈希值，去寻找对应的对象时，需要扫描所有的对象，计算对象的哈希值来对比查找。

3. BloomHitSet

类 BloomHitSet 采用压缩了的 Bloom filter 的方式来记录对象是否在缓存中。其原理实现在这里就不详细介绍了，它进一步减少了占用的内存空间。

特别需要注意的是：在模式 CACHEMODE_NONE（没有 cache tier）和模式 read only 模式下，是不需要 HitSet 来记录缓存命中的。只有在以下模式 write back、read forword、read_proxy 中才需要 HitSet 来记录缓存命中。

13.4.3　Cache Tier 的初始化

Cache Tier 初始化有两个入口，如下所示：

- on_active：如果该 pool 已经设置为 cache pool，在该 cache pool 的所有 PG 处于 activave 状态后初始化。
- on_pool_change：当该 pool 的所有 PG 都已经处于 active 状态后，才设置该 pool 为 cache pool，那么就等待 Monitor 通知 osd map 相关信息的变化，在 on_pool_change 函数里初始化。

1. hit_set_setup

函数 hit_set_setup 用来创建并初始化 HitSet 对象：

```
void ReplicatedPG::hit_set_setup()
```

具体实现如下：

1）首先检查如果 PG 既不处于 active 状态，也不处于 primary 状态，就调用函数 hit_set_clear 清除 hit_set 相关的记录，直接返回不做任何设置。

2）如果参数 pool.info.hit_set_count 为零，或者参数 pool.info.hit_set_period 为零，或者 pool.info.hit_set_params 参数为 HitSet::TYPE_NONE，就调用 hit_set_clear 清空 hit_set 的记录，并调用函数 hit_set_remove_all 删除所有的 HitSet 对象。

3）调用函数 hit_set_create 根据类型创建相应的 HitSet 类。

4）调用函数 hit_set_apply_log 来添加从 PG 日志中获取的新对象记录，并添加到 HitSet 中。

2. agent_setup

函数 agent_setup 完成 agent 相关的初始化工作：

```
void ReplicatedPG::agent_setup()
```

具体实现如下：

1）首先检查各种参数，如果 PG 处于下列情况之一：

- 不处于 active 状态。
- 非 primary。
- 模式是 pg_pool_t::CACHEMODE_NONE。
- pool.info.tier_of 没有设置。
- cache pool 不存在。

调用函数 agent_clear，停止 agent 线程，清除相关设置。

2）如果 agent_state 为空，就重新设置 agent_state 类，并设置相关的初始化参数。

3）调用函数 agent_choose_mode 设置 flush_mode 模式和 evict_mode 模式，并把 PG 添加到 agent 相应的工作队列中。

13.4.4 读写路径上的 Cache Tier 处理

在 OSD 的正常读写路径上，如果该 pool 有 Cache Tier 设置，处理逻辑就发生了变化。如下所示：

```
void ReplicatedPG::do_op(OpRequestRef& op){
{
  ......
  bool in_hit_set = false;
  if (hit_set) {
    if (obc.get()) {
      if (obc->obs.oi.soid != hobject_t() && hit_set->contains(obc->obs.oi.soid))
      in_hit_set = true;
    } else {
      if (missing_oid != hobject_t() && hit_set->contains(missing_oid))
        in_hit_set = true;
    }
    if (!op->hitset_inserted) {
      hit_set->insert(oid);
      op->hitset_inserted = true;
      if (hit_set->is_full() ||
          hit_set_start_stamp + pool.info.hit_set_period <= m->get_recv_stamp()) {
```

```
            hit_set_persist();
          }
        }
      }
      if (agent_state) {
        if (agent_choose_mode(false, op))
          return;
      }
      if (maybe_handle_cache(op,
                write_ordered,
                obc,
                r,
                missing_oid,
                false,
                in_hit_set))
        return;

      ......
    }
```

这里增加了相关的处理逻辑：

1）首先根据对象的上下文信息 obc 获取的情况，分两种情况调用函数 hit_set->contains 检查该对象是否命中，如果对象没有在 hit_set 中就调用 hit_set->insert 把该对象插入 hit_set 中。

2）如果 hit_set 已经满了，或者时间达到了 pool.info.hit_set_period 的值，就持久化保存该 hit_set 对象到磁盘上，创建一个新的 hit_set 对象。

3）调用函数 agent_choose_mode 对相应的 PG 设置 flush 模式和 evict 模式。

4）调用函数 maybe_handle_cache 来处理有关 cache 的读写请求。

下面分别介绍相应函数。

1. agent_choose_mode

函数 agent_choose_mode 用来计算一个 PG 的 flush_mode 和 evict_mode 的参数值：

```
bool ReplicatedPG::agent_choose_mode(bool restart, OpRequestRef op)
```

如果该函数返回值为 true，就表明该请求 Op 被重新加入请求队列里（由于 EvictMode 为 Full），其他情况都返回 false 值。

这个函数实现过程如下：

1）如果该 agent_state 处于 agent_state->delaying 状态，就返回 flase 值。

2）如果 info.stats.stats_invalid 为 true，表明当前的统计信息无效。agent 模式的计算依赖这些统计信息，在这种情况下暂时跳过，暂不计算 flush_mode 和 evict_mode 的值。

3）计算 divisor 值，也就是 cache_pool 里的 PG 数量。

4）计算 unflushable，也就是不能刷回的对象，并去掉 HitSet 相关的对象。

5）获取 base_pool 并检查如果 base_pool 不支持 omap，就去掉所有需要 omap 支持的对象。

6）计算 num_user_objects，其值为统计的对象数减去 unflushable 对象数。

7）计算 num_user_bytes，用统计信息的字节数减去 unflushable_bytes 的字节数。

8）计算脏对象的数目 num_dirty 的值，如果 base_pool 不支持 omap，就去掉带 omap 的对象。

9）计算 dirty_micro 和 full_micro，分别是脏数据的比率和数据满的比率，单位为百万分之一：

a）如果设置了 pool.info.target_max_bytes，就按照字节计算。首先计算每个对象的平均大小 avg_size：

- dirty_micro = 脏对象数目 × 每个对象的平均大小 / 每个 PG 的平均字节数 × 1000000
- full_micro = 用户对象数目 × 每个对象平均大小 / 每个 PG 的平均字节数 × 1000000

b）如果设置了 pool.info.target_max_objects，就按照对象数目来计算：

- dirty_objects_micro = 脏对象数目 / 每个 PG 的平均对象数目 × 1000000
- full_objects_micro = 用户对象数目 / 每个 PG 的平均对象数目 × 1000000

最终选择两种计算方式中最大的一个为最终结果。

10）获取 flush_target 和 flush_high_target 参数。如果是 restart，或者当前的 flush_mode 为 IDLE，就用 flush_slop 对二者增加，否则就减少（通过 flush_slop 对比率做修正）。

11）根据脏数据的比例，设置 flush_mode：

- 如果 dirty_micro 大于 flush_high_target，设置 flush_mode 为 TierAgentState::FLUSH_

MODE_HIGH。

- 如果 dirty-micro 大于 flush_target，就设置 flush_mode 为 TierAgentState::FLUSH_MODE_LOW。

12）获取 evict_target 的值，用 evict_slop 做一点比例的修正。

13）如果 full_micro 大于 1000000，evict_mode 就设置为 EVICT_MODE_FULL 模式，evict_effort 设置为最大 1000000。

14）如果 full_mircor 大于 evict_target，就设置 evict_mode 为 EVICT_MODE_SOME 类型。这时候需要计算 evict_effort 的值，evict_effort 为 PG 在 agent 工作队列里的优先级：

- over = full - evict_target
- span = 1 – evict_target
- evict_effort = over/span，转换成单位为百万分之一
- 通过 g_conf->osd_agent_quantize_effort 的修正，使得 evict_effort 的级别不会太多。

例 13-1　举例说明 evict_effort

例如：evict_target: 60%

full_micro : 80%

over = 80% - 60%

span = 1 – 60%

evict_effort = (80 %-60 %) / (1-60%) = 50%

本例子里为了便于理解，都使用了单位百分之一。

15）设置新计算的 flush_mode 值，并更新相关的统计信息。

16）设置新计算的 evict_mode 值，如果 evict_mode 是由 EVICT_MODE_FULL 变为其他类型，并且 PG 的状态为 is_active()，就需要把当前的 op 以及因 cache full 而等待的操作都重新加入请求队列，设置返回值为 true。

17）根据模式，做相应的处理：

a）如果 idle，就调用函数 agent_disable_pg 把该 PG 从 agent_queue 中删除。

b）如果是 retart，或者之前是 idle，就调用函数 agent_enable_pg 把该 PG 重新加入

agent_queue 处理队列中。

c）如果之前已经在队列中，就调用函数 agent_adjust_pg 来调整 evict_effort，也就是调整在队列中的优先级。

2. maybe_handle_cache_detail

函数 maybe_handle_cache 处理有关 cache 的逻辑，调用函数 maybe_handle_cache_detail 来完成，处理流程如下：

1）首先检查 op 的请求标志里如果有 CEPH_OSD_FLAG_IGNORE_CACHE 标志就直接返回。如果 pool.info.cache_mode 为 pg_pool_t::CACHEMODE_NONE 也直接返回。

2）如果可以获取 obc，并且是 write_ordered，该对象被阻塞，就返回 cache_result_t::NOOP 值。

3）如果该对象确实不存在，就返回 cache_result_t::NOOP 值。

4）如果可以获得 obc，并且对象存在，则认为缓存命中，则直接返回 cache_result_t::NOOP 值。

5）检查如果操作里设置了 op->need_skip_handle_cache()，就返回 cache_result_t::NOOP 值。

6）如果是 CACHEMODE_WRITEBACK 模式（下列情况都是缓存没有命中，cache pool 中没有需要的对象）：

情况 1：当前 agent_state 的 evict_mode 为 EVICT_MODE_FULL，说明当前的 cache 已经接近满了：

a）如果是只读操作：

```
if (!op->may_write() && !op->may_cache() &&!write_ordered &&
!must_promote)
```

如果可以 can_proxy_read，就调用函数 do_proxy_read 读取。否则调用 do_cache_redirect 函数返回给客户端 redirect 信息，客户端再去访问 base pool，这种方式就是 forward 方式读取。

b）如果是 write 操作，就调用 block_write_on_full_cache 阻塞当前的操作。

情况 2：不是 EVICT_MODE_FULL 模式：

a）如果 hit_set 为空：说明在这种模式下不需要 hitset（可能使 read only 模式），调用 promote_object 函数去 base pool 读取该对象。

b）如果 hit_set 不为空，并且 op->may_write() || op->may_cache()，也就是写操作：

- 如果允许 can_proxy_write，就调用 do_proxy_write 来以 proxy 的方式写入，否则调用 promote_object 操作，返回 cache_result_t::BLOCKED_PROMOTE
- 如果是以 proxy_write 方式写入，检查是否还需要调用函数 promote_object 操作。

c）否则，也就是 read 操作

- 如果允许 proxy_read，就调用函数 do_proxy_read 以代理的方式来读取。
- 检查如果需要 promote，就调用函数 maybe_promote 来检查。
- 如果既没有 promote，又没有 proxy_read，就调用 do_cache_redirect 来读取。

7）如果是 CACHEMODE_FORWARD 模式，就直接调用函数 do_cache_redirect，以 forward 模式读取。

8）如果是 CACHEMODE_READONLY 模式：如果是读操作，就调用 promote_object 函数从 base pool 读取；如果是写操作，就以 forward 的方式写。

9）如果是 CACHEMODE_READFORWARD 模式

- 如果是 write 操作，evict 的模式为 TierAgentState::EVICT_MODE_FULL，就阻塞，否则就调用函数 promote_object 来读取该对象，然后写入。
- 如果是读，就以 forward 模式读取。

如果是 CACHEMODE_READPROXY 模式，对于写，和 CACHEMODE_READFORWARD 处理相同；对于读，就以 proxy 模式读取。

下面介绍上述处理过程中的，处理代理读、代理写、从数据层加载对象到缓存层的具体处理函数。

函数 do_proxy_read 给 base pool 发送请求消息，来读取 cache pool 没有的对象数据。函数 finish_proxy_read 完成了 prox_read 的回调函数，其对结果做检查，并清理等待列表。通过函数 complete_read_ctx 给 client 端发送读操作的返回消息。特别注意的是，函数并没有把对象的数据写入 cache pool，所以后续还需要调用函数 promote_object 从 base pool 读

取该对象数据，然后写入 cache pool 中。

函数 do_proxy_write 直接写数据到 base pool 中。函数 finish_proxy_write 为 do_proxy_write 的回调函数，完成给客户端的发送 ACK 应对。同样，cache pool 中并没有该数据对象，还需要后续调用 promote_object 函数把对象从 base pool 中读到 cache pool 中。

函数 promote_object 完成 base pool 读取相关的对象并将其写入 cache pool 中，处理过程如下：

1）首先检查 scrubber.write_blocked_by_scrub 是否处于 block 状态。

2）构造 PromoteCallback 回调函数，然后调用函数 start_copy 拷贝数据。

函数 start_copy 用于执行 copy 操作拷贝数据，其处理过程如下：

1）检查如果 copy_ops 里有该对象，就调用函数 cancel_copy 取消正在进行的 copy 操作。

2）构造 CopyOp 操作，并添加到 copy_ops 的 map 中。

3）调用 obc->start_block()，把该 obc 设置为 blocked 为 true 的状态。

4）调用函数 _copy_some 做相应的操作。

函数 _copy_some 执行 copy 操作做部分数据量的拷贝，它是通过调用 osdc 的读操作来完成，其处理过程如下：

1）根据 cop->flags 设置各种 flags 标志。

2）构建 C_GatherBuilder，它是一个 Context 的回调函数。

3）调用函数 cop->cursor.is_initial() 判断是否为第一次操作，并且 cop->mirror_snapset 为 ture，就需要调用 list_snaps 获取 snapSet 的信息，并把结果保存在 cop->results.snapset 中，保存 tid 到 cop->objecter_tid2 中。

4）如果 cop->results.user_version 被设置了（在第一次操作中设置），在以后的操作中，需要执行 assert_version 操作。

5）如果该 pool 不支持 omap（ec），在 copyget_flags 标志中，设置 CEPH_OSD_COPY_GET_FLAG_NOTSUPP_OMAP 标志。

6）调用 op.copy_get 操作，设置相关的操作，并调用函数 op.set_last_op_flags 设置最后一个 op 的 flags 标志。

7）构造 C_Copyfrom 回调函数，调用函数 gather.set_finisher 设置 gather 的 finisher

函数。

8）调用函数 osd->objecter->read 把操作发送出去。

9）把 tid 设置为 fin->tid，调用函数 gather.activate() 使回调函数激活。

函数 process_copy_chunk 为 Copyfrom 的回调函数，完成了数据的部分写入。函数 finish_promote 为 PromoteCallback 的回调函数，完成后续 promote 操作。

13.4.5　cache 的 flush 和 evict 操作

cache pool 空间不够时，需要选择一些脏对象回刷到数据层（即 flush 操作），并将一些 clean 对象从缓存层剔除（即 evict 操作），以释放更多的缓存空间。这两种操作都是在后代完成的。flush 操作和 evict 操作算法的好坏，决定了 Cache Tier 的缓存命令率。

1. OSDService 中 Agent 相关的数据结构

在类 OSDService 中，定义了一个 AgentThread 的后台线程，用于完成两个任务：一是把 dirty 对象从 cache pool 层适时地回刷到 base pool 层，二是从 cache pool 层剔除掉一些不经常访问的 clean 对象。

```
Class OSDService{
  .....
  Mutex agent_lock;    //agent 线程锁，保护以下所有的数据结构
  Cond agent_cond;     // 线程相应的条件变量

  map<uint64_t, set<PGRef> > agent_queue;
  // 所有回刷或者剔除所需的 PG 集合，根据 PG 集合的优先级，保存在不同的 map 中

  set<PGRef>::iterator agent_queue_pos;
  // 当前在扫描的 PG 集合的一个位置，只有 agent_valid_iterator 为 true 时，这个指针才有效，
     否则从集合的起始处扫描
  bool agent_valid_iterator;

  int agent_ops;  // 所有正在进行的回刷和剔除操作
  int flush_mode_high_count;    // 如果回刷模式为 HIGH 模式，该该值就增加
  set<hobject_t, hobject_t::BitwiseComparator> agent_oids;
                        // 所有正在进行的 agent 的操作（回刷或剔除）的对象
  bool agent_active;     //agent 是否有效

  //agent 线程
   struct AgentThread : public Thread {
```

```
    OSDService *osd;
    explicit AgentThread(OSDService *o) : osd(o) {}
    void *entry() {
      osd->agent_entry();
      return NULL;
    }
  } agent_thread;

  bool agent_stop_flag;  //agent 停止的标志

  Mutex agent_timer_lock;
  SafeTimer agent_timer;
  //agent 相关的定时器，该定时器用于：当扫描一个 PG 对象时，该对象既没有剔除操作，也没有回刷
    操作，就停止 PG 的扫描，把该 PG 加入到定时器，5s 后继续
  ......
}
```

2. 数据结构 TierAgentState

在 ReplicatedPG 的内部，变量 agent_state 用来保存 PG 相关的 agent 信息。

```
struct TierAgentState {
  hobject_t position;    //PG 内扫描的对象位置
   int started;   // PG 里所有对象扫描完成后，所发起的所有 agent 操作数目。当 PG 扫描完成
                     所有的对象，如果没有 agent 操作，就需要延迟一段时间

  hobject_t start;   // 本次扫描起始位置
  bool delaying;     // 是否延迟

  // 历史的统计信息
  pow2_hist_t temp_hist;
  int hist_age;

  map<time_t,HitSetRef> hit_set_map; //HitSet 的历史记录

  // 最近处于 clean 的对象
  list<hobject_t> recent_clean;
}
```

3. agent_entry

agent_entry 是 agent_thread 的入口函数，它在后台完成 flush 操作和 evict 操作，具体处理流程如下：

1）加 agent_lock 锁。该锁保护 OSDService 里所有和 agent 相关的字段。

2）如果 agent_stop_flag 为 true，就将 agent_lock 解锁并退出，否则继续以下操作。

3）扫描 agent_queue 队列，如果 agent_queue 为空，就在条件变量 agent_cond 上等待。

4）从队列 agent_queue 中取出级别最高的 PG 的集合 top（始终从级别最高的取），如果 top 集合为空，就把该集合从 agent_queue 中删除，并使 agent_valid_iterator 集合内的 PG 指针设为 false，使其无效。

5）检查如果正在进行的 agent 操作数 agent_ops，如果该值大于所配置最大允许的 agent 操作的数量 g_conf->osd_agent_max_ops，或者非 agent_active，就等待。

6）如果 agent_valid_iterator 有效，就从 agent_queue_pos 处获取该 PG set 中的一个 PG；否则就从该 set 的头开始，获取相应的 PG 指针。

7）获取 agent 操作的最大数目 max 值，以及 agent_flush_quota 的值。

8）调用函数 pg->agent_work，正常情况返回 true 值。如果它的返回值为 false，该 PG 处于 delay 状态，需要加入 agent_timer 定时器，在配置项 g_conf->osd_agent_delay_time 设定的秒数后，调用 agent_choose_mode 重新设置模式。

4. agent_work

agent_work 完成一个 PG 内的 evict 操和 flush 操作，主要的流程如下：

1）调用函数 lock() 对 PG 加锁，agent_state 数据结构受它保护。

2）检查如果 agent_state 为空，或者 agent_state->is_idle()，就解锁返回 true 值。

3）调用函数 agent_load_hit_sets 加载 hit_set 的历史对象到内存中。

4）调用函数 pgbackend->objects_list_partial 来扫描本 PG 的对象，从 agent_state->position 开始位置扫描，结果保存在对象向量 ls 中。最小扫描一个对象，最大扫描数是配置参数 osd_pool_default_cache_max_evict_check_size 设置的对象个数。

5）对扫描到的 ls 中的对象，做如下检查：

- 跳过 hitset 对象。
- 跳过 degraded 对象。
- 跳过 missing 对象。
- 跳过 object context 不存在的对象。
- 跳过对象的 obs 不存储。
- 跳过正在进行 Scrub 操作的对象。

- 跳过已经被阻塞的对象。
- 跳过有正在读写请求的对象。
- 如果 base_pool 不支持 omap，就跳过有 omap 的对象。
- 如果 agent_state->evict_mode != TierAgentState::EVICT_MODE_IDLE，调用 agent_maybe_evict 函数来剔除掉一些对象。
- 如果当前的 flush_mode 不是 IDLE 状态，就调用 agent_maybe_flush 函数来回刷一些对象。
- 如果 started 大于 agent_max，也就是已经发起的 agent 操作数目大于最大允许的 agent_max 数目，就停止并退出。设置 next 的指针，也就是下次开始扫描的对象的起始位置。

6）更新 agent_state->hist_age 的值，如果 agent_state->hist_age 大于 g_conf->osd_agent_hist_halflife，就重置为 0，并调用函数 agent_state->temp_hist.decay() 把直方图的统计的对象减半。

7）比较扫描的指针，如果 PG 的对象扫描了一圈后，total_started 值为 0，也就是没有 agent 操作（flush 操作或者 evict 操作），就设置 need_delay 值为 true，需要延迟一段时间。

8）调用函数 hit_set_in_memory_trim 在内存中对一些旧的 hitset 对象做 trim 操作。

9）如果需要延迟，就调用函数 agent_delay 把该 PG 从 agent_queue 中删除。

10）调用 agent_choose_mode 重新计算 agent 的 evict 和 flush 的模式值。

5. agent_maybe_evict

函数 agent_maybe_evict 完成一个对象的 evict 操作，处理过程分析如下：

1）首先检查对象的状态：

- 如果不是 after_flush，就需要确保对象处于 clean 状态，否则直接返回 false 值。
- 如果该对象还有 watcher，说明还有客户端在使用该对象，返回 false 值。
- 对象如果处于 blocked 状态，返回 false 值。
- 如果对象处于 cache_pined 状态，返回 false 值。
- 验证对象的 clone 信息是否一致，如果不一致，返回 false 值。

2）如果 evict_mode 为 EVICT_MODE_SOME 模式：

- 需要确保该对象处于 clean 的时间大于 pool.info.cache_min_evict_age 的长度。
- 调用函数 agent_estimate_temp 计算该对象的热度值 temp。
- 调用函数 get_position_micro 计算该对象的 temp 值在对象的统计直方图的位置值 emp_upper。
- 如果 1000000 减去 temp_upper 的值大于等于 agent_state->evict_effort，该对象就不删除，直接返回 false 值。

3）如果 evict_mode 为 EVICT_MODE_FULL 模式，就调用函数 _delete_oid 删除该对象，最后调用函数 simple_opc_submit 发起实践的删除请求。

6. agent_maybe_flush

函数 agent_maybe_flush 完成一个对象的 flush 操作，处理过程如下：

1）检查如果对象不脏，就返回 false 值。

2）检查如果该对象处于 cache_pinned，就返回 false 值。

3）如果当前 evict_mode 不是 evict_mode_full 状态，就检查该对象处于 dirty 的时间是否超过 pool.info.cache_min_flush_age 值。

4）检查对象是否处于 activate 状态。

5）调用函数 start_flush 来完成对象的回刷操作。

7. start_flush

函数 start_flush 完成实际的 flush 操作，具体处理过程如下：

1）首先调用 obc->ssc->snapset.get_filtered 把已经删除的 snap 对象过滤掉。

2）检查比当前 clone 对象版本更早的克隆对象：

- 如果还有版本更早的克隆对象处于 missing 状态，就返回 -ENOENT 错误码。
- 如果还有版本更早的克隆对象，获取该对象的 older_obc 值，并且标记该对象处于 dirty 状态，直接返回 EBUSY；如果该对象的 obc 不存在，就默认为 clean 状态。

例如：clones[1,4,5,8]，当前 snap 为 5，就必须确保 [1, 4] 处于 clean 状态。

3）如果 blocking 设置为 true，就设置对象处于 blocked 状态。

4）检查如果该对象在 flush_ops 中，也就是该对象已经在 flush 中，根据不同的情况分别处理。

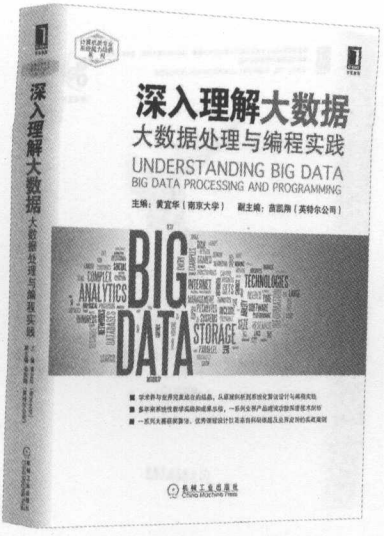
5）最后处理克隆对象的 flush 操作。

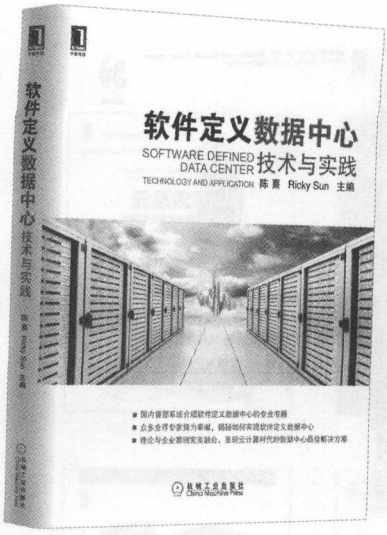
13.5 本章小结

本章介绍了 Ceph 的 Cache Tier 的原理和架构，及其各种模式，以及因引入 Cache Tier 而引起 Ceph 的读写路径上不同的处理。最后介绍了 Caceh Tier 后台的 flush 操作和 evict 操作的处理过程。

Ceph 的 Cache Tier 功能目前在对象的访问频率和热点统计上的实现都比较简单，有待后续实现更好的基于自学习的 Cache 算法。

推荐阅读





深入理解大数据：大数据处理与编程实践

作者：黄宜华 等 ISBN：978-7-111-47325-1 定价：79.00元

本书在总结多年来MapReduce并行处理技术课程教学经验和成果的基础上，与业界著名企业Intel公司的大数据技术和产品开发团队和资深工程师联合，以学术界的教学成果与业界高水平系统研发经验完美结合，在理论联系实际的基础上，在基础理论原理、实际算法设计方法以及业界深度技术三个层面上，精心组织材料编写而成。

作为国内第一本经过多年课堂教学实践总结而成的大数据并行处理和编程技术书籍，本书全面地介绍了大数据处理相关的基本概念和原理，着重讲述了Hadoop MapReduce大数据处理系统的组成结构、工作原理和编程模型，分析了基于MapReduce的各种大数据并行处理算法和程序设计的思想方法。适合高等院校作为MapReduce大数据并行处理技术课程的教材，同时也很适合作为大数据处理应用开发和编程专业技术人员的参考手册。

——中国工程院院士、中国计算机学会大数据专家委员会主任 李国杰

软件定义数据中心——技术与实践

作者：陈熹 孙宇熙 ISBN：978-7-111-48317-5 定价：69.00元

国内首部系统介绍软件定义数据中心的专业书籍。

众多业界专家倾力奉献，揭秘如何实现软件定义数据中心。

理论与企业案例完美融合，呈现云计算时代的数据中心最佳解决方案。

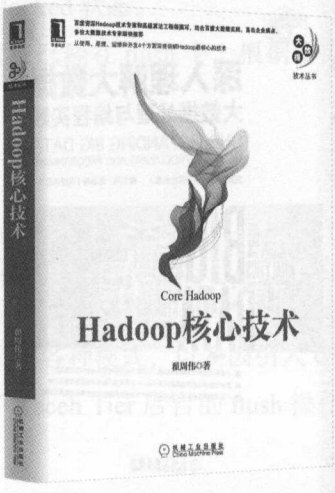
有了以软件定义数据中心为基础的混合云，企业就可以进退有度，游刃有余，加上成功管理新的移动终端技术，可轻松进入“云移动”时代！这也是为什么软件定义数据中心最近获得大家注意的根本原因。EMC中国研究院编著的这本《软件定义数据中心：技术与实践》恰逢其时，它会给读者详细解说怎么实现软件定义数据中心。

——VMware高级副总裁，EMC中国卓越研发集团创始人 Charles Fan

推荐阅读

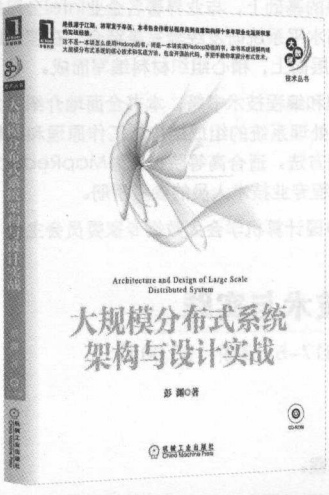
Spark大数据处理：技术、应用与性能优化

作者：高彦杰 ISBN：978-7-111-48386-1 定价：59.00元



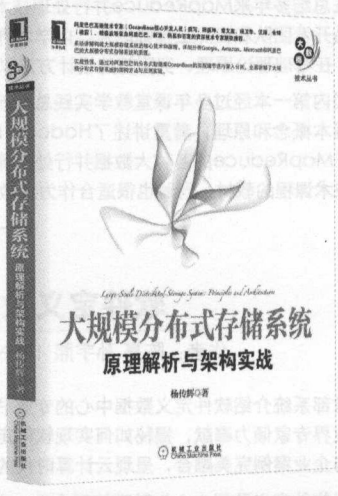
Hadoop核心技术

作者：翟周伟 ISBN：978-7-111-49468-3 定价：69.00元



大规模分布式系统架构与设计实战

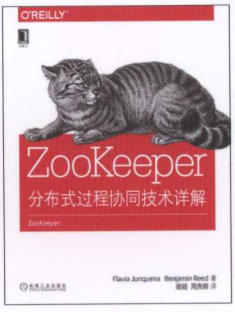
作者：彭渊 ISBN：978-7-111-45503-5 定价：59.00元

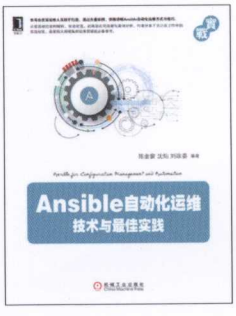


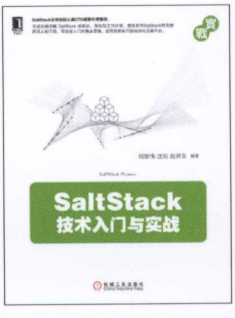
大规模分布式存储系统：原理解析与架构实战

作者：杨传辉 ISBN：978-7-111-43052-0 定价：59.00元

推荐阅读







封面设计 · 王建敏

这本书是目前我所看到的从代码角度解读Ceph的最好作品，即使在全球范围内，都没有类似的书籍能够与之媲美。信每个Ceph爱好者都能从这本书中找到自己心中某些疑问的解答途径。

—— **王豪迈** XSKY公司C

看到常涛写的这本书很是欣慰，很高兴国内有了一本专业书供相关人员去接触和学习Ceph代码，这有助于增加国内发者在Ceph社区的代码提交量。另外Ceph中国社区也将计划组织Ceph Hackathon活动，以此来促进Ceph代码的性循环，取之于Ceph社区，回馈于Ceph社区。我们要让全世界开发者看到中国人不仅仅在OpenStack中贡献大，Ceph社区中的贡献同样显著。

—— **耿航** Ceph中国社区联合创始人，XSKY公司市场技术专

本书主要内容包括

- Ceph数据读写功能关键流程的实现。
- Ceph如何做到数据强一致性，并可大规模部署。
- Ceph数据分布式算法CRUSH如何实现只通过计算就可定位数据块。
- Ceph如何实现存储的高级功能：如快照、克隆、纠删码、Scrub、Cache Tier等。
- Ceph数据修复的实现过程，包括Recovery和Backfill两个过程的具体实现。
- Ceph如何实现集群伸缩时的数据迁移和数据均衡。

上架指导：计算机/云计算
ISBN 978-7-111-55207-9
9 787111 552079
定价：59.00元

投稿热线：(010) 88379604
客服热线：(010) 88379426 88361066
购书热线：(010) 68326294 88379649 68995259

华章网站：www.hzbook.com
网上购书：www.china-pub.com
数字阅读：www.hzmedia.com.cn